spikingjelly

发布 *alpha*

PKU MLG, PCL

2021 年 03 月 25 日

上手教程

1 安装 3

2 模块文档 105

3 文档索引 107

4 引用 109

5 项目信息 111

6 Welcome to SpikingJelly’s documentation 113

Python 模块索引 333

索引 335

SpikingJelly 是一个基于 PyTorch，使用脉冲神经网络 (Spiking Neural Network, SNN) 进行深度学习的框架。

- *Homepage in English*

CHAPTER 1

安装

注意，SpikingJelly 是基于 PyTorch 的，需要确保环境中已经安装了 PyTorch，才能安装 spikingjelly。

从 PyPI 安装：

```
pip install spikingjelly
```

PyPI 的安装包不包含 CUDA 扩展。如果想使用 CUDA 扩展，请 **从源代码安装**：

通过 GitHub：

```
git clone https://github.com/fangwei123456/spikingjelly.git
cd spikingjelly
python setup.py install
```

通过 OpenI：

```
git clone https://git.openi.org.cn/OpenI/spikingjelly.git
cd spikingjelly
python setup.py install
```

1.1 时间驱动

本教程作者：fangwei123456

本节教程主要关注 `spikingjelly.clock_driven`，介绍时钟驱动的仿真方法、梯度替代法的概念、可微分 SNN 神经元的使用方式。

梯度替代法是近年来兴起的一种新方法，关于这种方法的更多信息，可以参见如下综述：

Neftci E, Mostafa H, Zenke F, et al. Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-based optimization to spiking neural networks[J]. IEEE Signal Processing Magazine, 2019, 36(6): 51-63.

此文的下载地址可以在 arXiv 上找到。

1.1.1 SNN 之于 RNN

可以将 SNN 中的神经元看作是一种 RNN，它的输入是电压增量（或者是电流与膜电阻的乘积，但为了方便，在 `clock_driven.neuron` 中用电压增量），隐藏状态是膜电压，输出是脉冲。这样的 SNN 神经元是具有马尔可夫性的：当前时刻的输出只与当前时刻的输入、神经元自身的状态有关。

可以用充电、放电、重置，这 3 个离散方程来描述任意的离散脉冲神经元：

$$
\begin{aligned}
H(t) &= f(V(t-1), X(t)) \\
S(t) &= g(H(t) - V_{threshold}) = \Theta(H(t) - V_{threshold}) \\
V(t) &= H(t) \cdot (1 - S(t)) + V_{reset} \cdot S(t)
\end{aligned}
$$

其中 $V(t)$ 是神经元的膜电压；$X(t)$ 是外源输入，例如电压增量；$H(t)$ 是神经元的隐藏状态，可以理解为神经元还没有发放脉冲前的瞬时；$f(V(t-1), X(t))$ 是神经元的状态更新方程，不同的神经元，区别就在于更新方程不同。

例如对于 LIF 神经元，描述其阈下动态的微分方程，以及对应的差分方程为：

$$
\begin{aligned}
\tau_m \frac{\mathrm{d}V(t)}{\mathrm{d}t} &= -(V(t) - V_{reset}) + X(t) \\
\tau_m (V(t) - V(t-1)) &= -(V(t-1) - V_{reset}) + X(t)
\end{aligned}
$$

对应的充电方程为

$$
f(V(t-1), X(t)) = V(t-1) + \frac{1}{\tau_m}(-(V(t-1) - V_{reset}) + X(t))
$$

放电方程中的 $S(t)$ 是神经元发放的脉冲，$g(x) = \Theta(x)$ 是阶跃函数，RNN 中习惯称之为门控函数，我们在 SNN 中则称呼其为脉冲函数。脉冲函数的输出仅为 0 或 1，可以表示脉冲的发放过程，定义为

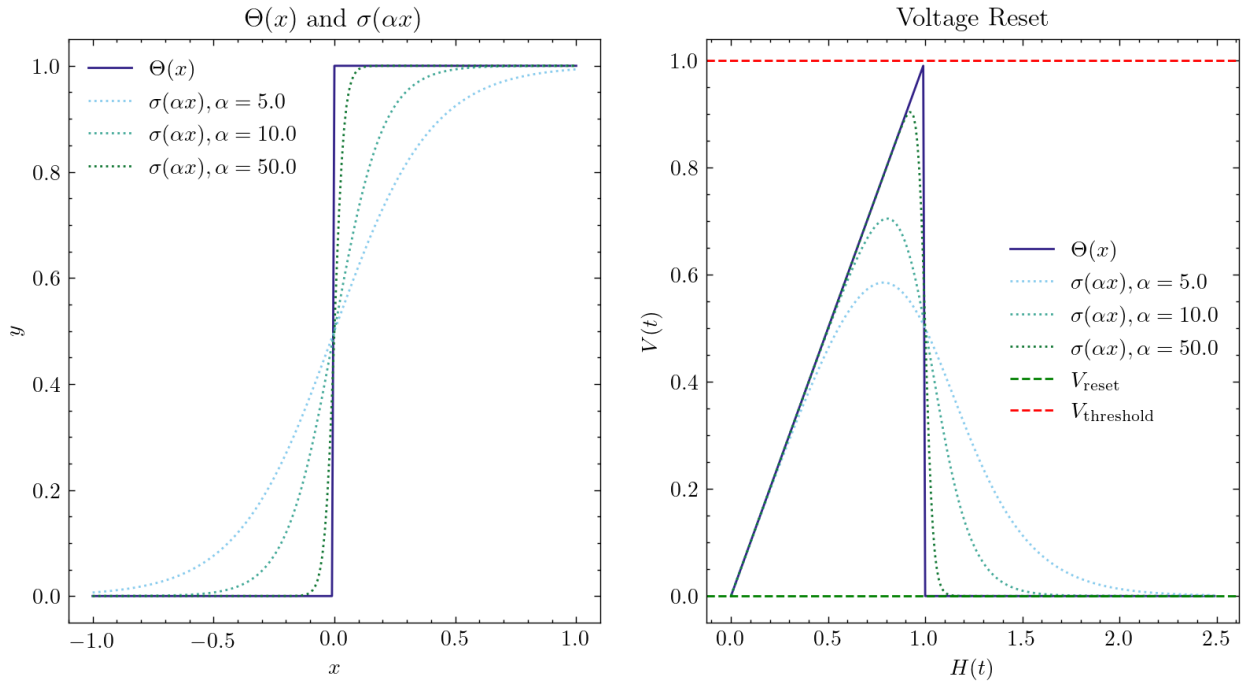
$$
\Theta(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}
$$

重置表示电压重置过程：发放脉冲，则电压重置为 V_{reset}；没有发放脉冲，则电压不变。

1.1.2 梯度替代法

RNN 使用可微分的门控函数，例如 tanh 函数。而 SNN 的脉冲函数 $g(x) = \Theta(x)$ 显然是不可微分的，这就导致了 SNN 虽然一定程度上与 RNN 非常相似，但不能用梯度下降、反向传播来训练。但我们可以用一个形状与 $g(x) = \Theta(x)$ 非常相似，但可微分的门控函数 $\sigma(x)$ 去替换它。

这一方法的核心思想是：在前向传播时，使用 $g(x) = \Theta(x)$，神经元的输出是离散的 0 和 1，我们的网络仍然是 SNN；而反向传播时，使用梯度替代函数的梯度 $g'(x) = \sigma'(x)$ 来代替脉冲函数的梯度。最常见的梯度替代函数即为 sigmoid 函数 $\sigma(\alpha x) = \frac{1}{1+exp(-\alpha x)}$，$\alpha$ 可以控制函数的平滑程度，越大的 α 会越逼近 $\Theta(x)$ 但越容易在靠近 $x = 0$ 时梯度爆炸，远离 $x = 0$ 时则容易梯度消失，导致网络也会越难以训练。下图显示了不同的 α 时，梯度替代函数的形状，以及对应的重置方程的形状：



默认的梯度替代函数为 `clock_driven.surrogate.Sigmoid()`，在 `clock_driven.surrogate` 中还提供了其他的可选近似门控函数。梯度替代函数是 `clock_driven.neuron` 中神经元构造函数的参数之一：

```
class BaseNode(nn.Module):
    def __init__(self, v_threshold=1.0, v_reset=0.0, surrogate_function=surrogate.
↪Sigmoid(), monitor_state=False):
        '''
        :param v_threshold: 神经元的阈值电压
        :param v_reset: 神经元的重置电压。如果不为 None，当神经元释放脉冲后，电压会被重置为 v_
↪reset；如果设置为 None，则电压会被减去阈值
        :param surrogate_function: 反向传播时用来计算脉冲函数梯度的替代函数
        :param monitor_state: 是否设置监视器来保存神经元的电压和释放的脉冲。
                              若为 True，则 self.monitor 是一个字典，键包括'v'和's'，分别记录电压和输
出脉冲。对应的值是一个链表。为了节省显存（内存），列表中存入的是原始变量
```

（下页继续）

(续上页)

```
                    转换为 numpy 数组后的值。还需要注意，self.reset() 函数会清空这些链表
```

如果想要自定义新的近似门控函数，可以参考 `clock_driven.surrogate` 中的代码实现。通常是定义 `torch.autograd.Function`，然后将其封装成一个 `torch.nn.Module` 的子类。

1.1.3 将脉冲神经元嵌入到深度网络

解决了脉冲神经元的微分问题后，我们的脉冲神经元可以像激活函数那样，嵌入到使用 PyTorch 搭建的任意网络中，使得网络成为一个 SNN。在 `clock_driven.neuron` 中已经实现了一些经典神经元，可以很方便地搭建各种网络，例如一个简单的全连接网络：

```
net = nn.Sequential(
        nn.Linear(100, 10, bias=False),
        neuron.LIFNode(tau=100.0, v_threshold=1.0, v_reset=5.0)
        )
```

1.1.4 使用双层全连接网络进行 MNIST 分类

现在我们使用 `clock_driven.neuron` 中的 LIF 神经元，搭建一个双层全连接网络，对 MNIST 数据集进行分类。

首先定义我们的网络结构：

```
class Net(nn.Module):
    def __init__(self, tau=100.0, v_threshold=1.0, v_reset=0.0):
        super().__init__()
        # 网络结构，简单的双层全连接网络，每一层之后都是 LIF 神经元
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28 * 28, 14 * 14, bias=False),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset),
            nn.Linear(14 * 14, 10, bias=False),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset)
        )

    def forward(self, x):
        return self.fc(x)
```

定义我们的超参数：

```
device = input('输入运行的设备，例如“cpu”或“cuda:0” \n input device, e.g., "cpu" or
↪"cuda:0": ')
dataset_dir = input('输入保存 MNIST 数据集的位置，例如“./” \n input root directory for␣
↪saving MNIST dataset, e.g., "./": ')
batch_size = int(input('输入 batch_size，例如“64” \n input batch_size, e.g., "64": '))
learning_rate = float(input('输入学习率，例如“1e-3” \n input learning rate, e.g., "1e-3
↪": '))
T = int(input('输入仿真时长，例如“100” \n input simulating steps, e.g., "100": '))
tau = float(input('输入 LIF 神经元的时间常数 tau，例如“100.0” \n input membrane time␣
↪constant, tau, for LIF neurons, e.g., "100.0": '))
train_epoch = int(input('输入训练轮数，即遍历训练集的次数，例如“100” \n input training␣
↪epochs, e.g., "100": '))
log_dir = input('输入保存 tensorboard 日志文件的位置，例如“./” \n input root directory for␣
↪saving tensorboard logs, e.g., "./": ')
```

初始化数据加载器、网络、优化器，以及编码器（我们使用泊松编码器，将 MNIST 图像编码成脉冲序列）：

```
# 初始化网络
net = Net(tau=tau).to(device)
# 使用 Adam 优化器
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
# 使用泊松编码器
encoder = encoding.PoissonEncoder()
```

网络的训练很简单。将网络运行 `T` 个时间步长，对输出层 10 个神经元的输出脉冲进行累加，得到输出层脉冲释放次数 `out_spikes_counter`；使用脉冲释放次数除以仿真时长，得到输出层脉冲发放频率 `out_spikes_counter_frequency = out_spikes_counter / T`。我们希望当输入图像的实际类别是 `i` 时，输出层中第 `i` 个神经元有最大的激活程度，而其他神经元都保持沉默。因此损失函数自然定义为输出层脉冲发放频率 `out_spikes_counter_frequency` 与实际类别进行 one hot 编码后得到的 `label_one_hot` 的交叉熵，或 MSE。我们使用 MSE，因为实验发现 MSE 的效果更好一些。尤其需要注意的是，SNN 是有状态，或者说有记忆的网络，因此在输入新数据前，一定要将网络的状态重置，这可以通过调用 `clock_driven.functional.reset_net(net)` 来实现。训练的代码如下：

```
for img, label in train_data_loader:
    img = img.to(device)
    label = label.to(device)
    label_one_hot = F.one_hot(label, 10).float()

    optimizer.zero_grad()

    # 运行 T 个时长，out_spikes_counter 是 shape=[batch_size, 10] 的 tensor
    # 记录整个仿真时长内，输出层的 10 个神经元的脉冲发放次数
    for t in range(T):
        if t == 0:
```

（下页继续）

(续上页)

```
            out_spikes_counter = net(encoder(img).float())
        else:
            out_spikes_counter += net(encoder(img).float())

    # out_spikes_counter / T 得到输出层 10 个神经元在仿真时长内的脉冲发放频率
    out_spikes_counter_frequency = out_spikes_counter / T

    # 损失函数为输出层神经元的脉冲发放频率，与真实类别的 MSE
    # 这样的损失函数会使，当类别 i 输入时，输出层中第 i 个神经元的脉冲发放频率趋近 1，而其他神经元的脉
冲发放频率趋近 0
    loss = F.mse_loss(out_spikes_counter_frequency, label_one_hot)
    loss.backward()
    optimizer.step()
    # 优化一次参数后，需要重置网络的状态，因为 SNN 的神经元是有“记忆”的
    functional.reset_net(net)
```

测试的代码与训练代码相比更为简单：

```
net.eval()
with torch.no_grad():
    # 每遍历一次全部数据集，就在测试集上测试一次
    test_sum = 0
    correct_sum = 0
    for img, label in test_data_loader:
        img = img.to(device)
        for t in range(T):
            if t == 0:
                out_spikes_counter = net(encoder(img).float())
            else:
                out_spikes_counter += net(encoder(img).float())

        correct_sum += (out_spikes_counter.max(1)[1] == label.to(device)).float().
↪sum().item()
        test_sum += label.numel()
        functional.reset_net(net)

    writer.add_scalar('test_accuracy', correct_sum / test_sum, epoch)
```

完整的代码位于 `clock_driven.examples.lif_fc_mnist.py`，在代码中我们还使用了 Tensorboard 来保存训练日志。可以直接在 Python 命令行运行它：

```
>>> import spikingjelly.clock_driven.examples.lif_fc_mnist as lif_fc_mnist
>>> lif_fc_mnist.main()
输入运行的设备，例如“cpu”或“cuda:0”
```

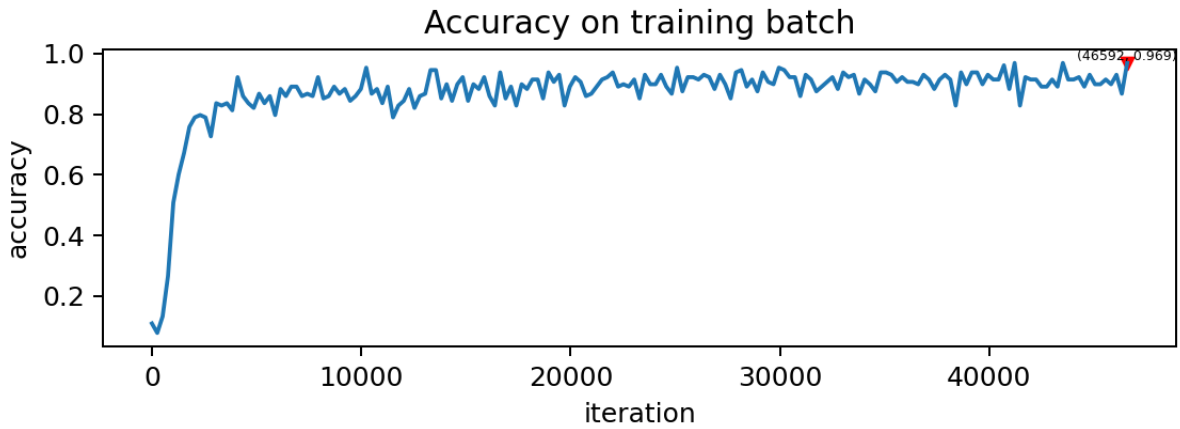
(下页继续)

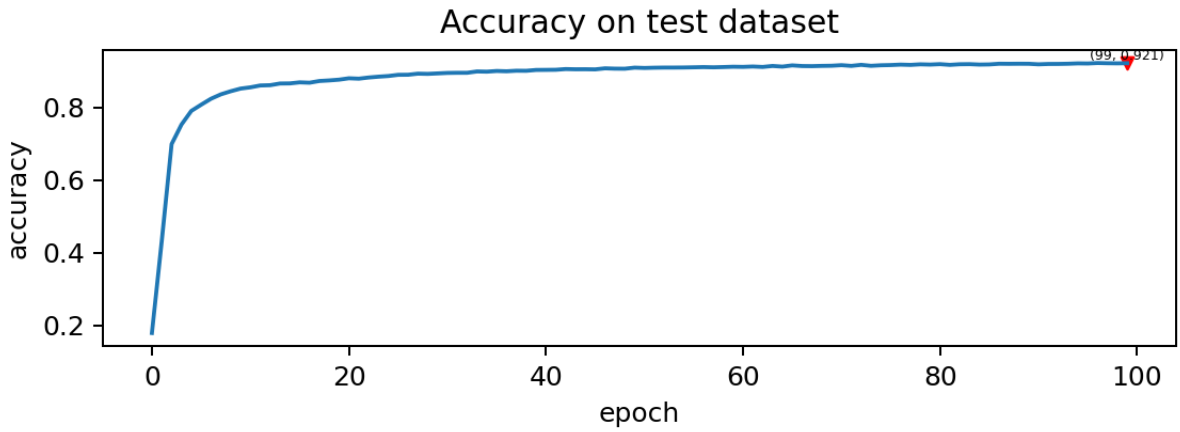
(续上页)

```
 input device, e.g., "cpu" or "cuda:0": cuda:15
输入保存 MNIST 数据集的位置，例如“./”
 input root directory for saving MNIST dataset, e.g., "./": ./mnist
输入 batch_size，例如“64”
 input batch_size, e.g., "64": 128
输入学习率，例如“1e-3”
 input learning rate, e.g., "1e-3": 1e-3
输入仿真时长，例如“100”
 input simulating steps, e.g., "100": 50
输入 LIF 神经元的时间常数 tau，例如“100.0”
 input membrane time constant, tau, for LIF neurons, e.g., "100.0": 100.0
输入训练轮数，即遍历训练集的次数，例如“100”
 input training epochs, e.g., "100": 100
输入保存 tensorboard 日志文件的位置，例如“./”
 input root directory for saving tensorboard logs, e.g., "./": ./logs_lif_fc_
↪mnist
cuda:15 ./mnist 128 0.001 50 100.0 100 ./logs_lif_fc_mnist
train_times 0 train_accuracy 0.109375
cuda:15 ./mnist 128 0.001 50 100.0 100 ./logs_lif_fc_mnist
train_times 1024 train_accuracy 0.5078125
cuda:15 ./mnist 128 0.001 50 100.0 100 ./logs_lif_fc_mnist
train_times 2048 train_accuracy 0.7890625
...
cuda:15 ./mnist 128 0.001 50 100.0 100 ./logs_lif_fc_mnist
train_times 46080 train_accuracy 0.9296875
```

需要注意的是，训练这样的 SNN，所需显存数量与仿真时长 `T` 线性相关，更长的 `T` 相当于使用更小的仿真步长，训练更为“精细”，但训练效果不一定更好，因此 `T` 太大，SNN 在时间上展开后就会变成一个非常深的网络，梯度的传递容易衰减或爆炸。由于我们使用了泊松编码器，因此需要较大的 `T`。

我们的这个模型，在 Tesla K80 上训练 100 个 epoch，大约需要 75 分钟。训练时每个 batch 的正确率、测试集正确率的变化情况如下：





最终达到大约 92% 的测试集正确率，这并不是一个很高的正确率，因为我们使用了非常简单的网络结构，以及泊松编码器。我们完全可以去掉泊松编码器，将图像直接送入 SNN，在这种情况下，首层 LIF 神经元可以被视为编码器。

1.2 事件驱动

本教程作者：fangwei123456

本节教程主要关注 `spikingjelly.event_driven`，介绍事件驱动概念、Tempotron 神经元。

1.2.1 事件驱动的 SNN 仿真

`clock_driven` 使用时间驱动的方法对 SNN 进行仿真，因此在代码中都能够找到在时间上的循环，例如：

```
for t in range(T):
    if t == 0:
        out_spikes_counter = net(encoder(img).float())
```

(下页继续)

(续上页)

```
    else:
        out_spikes_counter += net(encoder(img).float())
```

而使用事件驱动的 SNN 仿真，并不需要在时间上进行循环，神经元的状态更新由事件触发，例如产生脉冲或接受输入脉冲，因而不同神经元的活动可以异步计算，不需要在时钟上保持同步。

1.2.2 脉冲响应模型 (Spike response model, SRM)

在脉冲响应模型 (Spike response model, SRM) 中，使用显式的 $V-t$ 方程来描述神经元的活动，而不是用微分方程去描述神经元的充电过程。由于 $V-t$ 是已知的，因此给与任何输入 $X(t)$，神经元的响应 $V(t)$ 都可以被直接算出。

1.2.3 Tempotron 神经元

Tempotron 神经元是[1] 提出的一种 SNN 神经元，其命名来源于 ANN 中的感知器（Perceptron）。感知器是最简单的 ANN 神经元，对输入数据进行加权求和，输出二值 0 或 1 来表示数据的分类结果。Tempotron 可以看作是 SNN 领域的感知器，它同样对输入数据进行加权求和，并输出二分类的结果。

Tempotron 的膜电位定义为：

$$V(t) = \sum_i w_i \sum_{t_i} K(t-t_i) + V_{reset}$$

其中 w_i 是第 i 个输入的权重，也可以看作是所连接的突触的权重；t_i 是第 i 个输入的脉冲发放时刻，$K(t-t_i)$ 是由于输入脉冲引发的突触后膜电位 (postsynaptic potentials, PSPs)；V_{reset} 是 Tempotron 的重置电位，或者叫静息电位。

$K(t-t_i)$ 是一个关于 t_i 的函数 (PSP Kernel),[?] 中使用的函数形式如下：

$$K(t-t_i) = \begin{cases} V_0(exp(-\frac{t-t_i}{\tau}) - exp(-\frac{t-t_i}{\tau_s})), & t \geq t_i \\ 0, & t < t_i \end{cases}$$

其中 V_0 是归一化系数，使得函数的最大值为 1；τ 是膜电位时间常数，可以看出输入的脉冲在 Tempotron 上会引起瞬时的点位激增，但之后会指数衰减；τ_s 则是突触电流的时间常数，这一项的存在表示突触上传导的电流也会随着时间衰减。

单个的 Tempotron 可以作为一个二分类器，分类结果的判别，是看 Tempotron 的膜电位在仿真周期内是否过阈值:

$$y = \begin{cases} 1, & V_{t_{max}} \geq V_{threshold} \\ 0, & V_{t_{max}} < V_{threshold} \end{cases}$$

其中 $t_{max} = \text{argmax}\{V_t\}$。从 Tempotron 的输出结果也能看出，Tempotron 只能发放不超过 1 个脉冲。单个 Tempotron 只能做二分类，但多个 Tempotron 就可以做多分类。

[1] Gutig R, Sompolinsky H. The tempotron: a neuron that learns spike timing–based decisions[J]. Nature Neuroscience, 2006, 9(3): 420-428.

1.2.4 如何训练 Tempotron

使用 Tempotron 的 SNN 网络，通常是“全连接层 + Tempotron”的形式，网络的参数即为全连接层的权重。使用梯度下降法来优化网络参数。

以二分类为例，损失函数被定义为仅在分类错误的情况下存在。当实际类别是 1 而实际输出是 0，损失为 $V_{threshold} - V_{t_{max}}$；当实际类别是 0 而实际输出是 1，损失为 $V_{t_{max}} - V_{threshold}$。可以统一写为：

$$E = (y - \hat{y})(V_{threshold} - V_{t_{max}})$$

直接对参数求梯度，可以得到：

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= (y - \hat{y})(\sum_{t_i < t_{max}} K(t_{max} - t_i) + \frac{\partial V(t_{max})}{\partial t_{max}} \frac{\partial t_{max}}{\partial w_i}) \\ &= (y - \hat{y})(\sum_{t_i < t_{max}} K(t_{max} - t_i))\end{aligned}$$

因为 $\frac{\partial V(t_{max})}{\partial t_{max}} = 0$。

1.2.5 并行实现

如前所述，对于脉冲响应模型，一旦输入给定，神经元的响应方程已知，任意时刻的神经元状态都可以求解。此外，计算 t 时刻的电压值，并不需要依赖于 $t-1$ 时刻的电压值，因此不同时刻的电压值完全可以并行求解。在 `spikingjelly/event_driven/neuron.py` 中实现了集成全连接层、并行计算的 Tempotron，将时间看作是一个单独的维度，整个网络在 $t = 0, 1, ..., T-1$ 时刻的状态全都被并行地计算出。读者如有兴趣可以直接阅读源代码。

1.2.6 识别 MNIST

我们使用 Tempotron 搭建一个简单的 SNN 网络，识别 MNIST 数据集。首先我们需要考虑如何将 MNIST 数据集转化为脉冲输入。在“clock_driven“中的泊松编码器，在伴随着整个网络的 for 循环中，不断地生成脉冲；但在使用 Tempotron 时，我们使用高斯调谐曲线编码器[2]，这一编码器可以在时间维度上并行地将输入数据转化为脉冲发放时刻。

高斯调谐曲线编码器

假设我们要编码的数据有 n 个特征，对于 MNIST 图像，因其是单通道图像，可以认为 $n = 1$。高斯调谐曲线编码器，使用 $m(m > 2)$ 个神经元去编码每个特征，并将每个特征编码成这 m 个神经元的脉冲发放时刻，因此可以认为编码器内共有 nm 个神经元。

[2] Bohte S M, Kok J N, La Poutre J A, et al. Error-backpropagation in temporally encoded networks of spiking neurons[J]. Neurocomputing, 2002, 48(1): 17-37.

对于第 i 个特征 X^i，它的取值范围为 $X^i_{min} \le X^i \le X^i_{max}$，首先计算出 m 条高斯曲线 g^i_j 的均值和方差：

$$\mu^i_j = x^i_{min} + \frac{2j-3}{2}\frac{x^i_{max}-x^i_{min}}{m-2}, j = 1, 2, ..., m$$
$$\sigma^i_j = \frac{1}{\beta}\frac{x^i_{max}-x^i_{min}}{m-2}$$

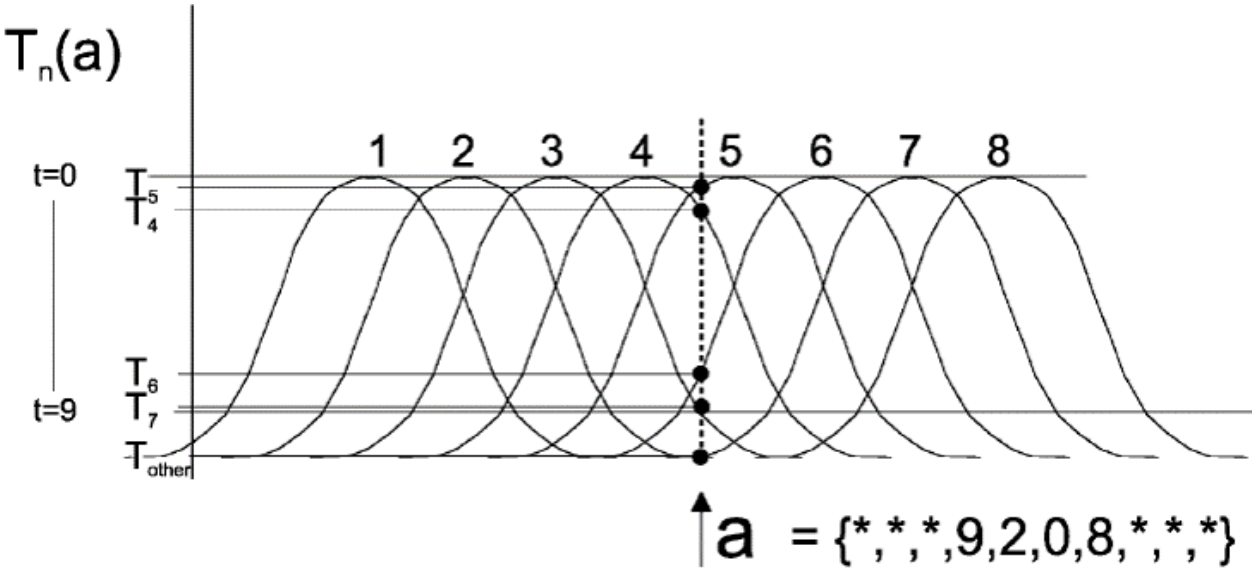
其中 β 通常取值为 1.5。可以看出，这 m 条高斯曲线的形状完全相同，只是对称轴所在的位置不同。

对于要编码的数据 $x \in X^i$，首先计算出 x 对应的高斯函数值 $g^i_j(x)$，这些函数值全部介于 $[0, 1]$ 之间。接下来，将函数值线性地转换到 $[0, T]$ 之间的脉冲发放时刻，其中 T 是编码周期，或者说是仿真时长：

$$t_j = \text{Round}((1 - g^i_j(x))T)$$

其中 Round 取整函数。此外，对于发放时刻太晚的脉冲，例如发放时刻为 T，则直接将发放时刻设置为 -1，表示没有脉冲发放。

形象化的示例如下图[7] 所示，要编码的数据 $x \in X^i$ 是一条垂直于横轴的直线，与 m 条高斯曲线相交于 m 个交点，这些交点在纵轴上的投影点，即为 m 个神经元的脉冲发放时刻。但由于我们在仿真时，仿真步长通常是整数，因此脉冲发放时刻也需要取整。



定义网络、损失函数、分类结果

网络的结构非常简单，单层的 Tempotron，输出层是 10 个神经元，因为 MNIST 图像共有 10 类：

```
class Net(nn.Module):
    def __init__(self, m, T):
        # m 是高斯调谐曲线编码器编码一个像素点所使用的神经元数量
        super().__init__()
        self.tempotron = neuron.Tempotron(784*m, 10, T)
```

(下页继续)

(续上页)

```
    def forward(self, x: torch.Tensor):
        # 返回的是输出层 10 个 Tempotron 在仿真时长内的电压峰值
        return self.tempotron(x, 'v_max')
```

分类结果被认为是输出的 10 个电压峰值的最大值对应的神经元索引，因此训练时正确率计算如下：

```
train_acc = (v_max.argmax(dim=1) == label.to(device)).float().mean().item()
```

我们使用的损失函数与[?] 中的类似，但所有不同。对于分类错误的神经元，误差为其峰值电压与阈值电压之差的平方，损失函数可以在 `event_driven.neuron` 中找到源代码：

```
class Tempotron(nn.Module):
    ...
    @staticmethod
    def mse_loss(v_max, v_threshold, label, num_classes):
        '''
        :param v_max: Tempotron 神经元在仿真周期内输出的最大电压值，与 forward 函数在 ret_
↪type == 'v_max' 时的返回值相\
        同。shape=[batch_size, out_features] 的 tensor
        :param v_threshold: Tempotron 的阈值电压，float 或 shape=[batch_size, out_
↪features] 的 tensor
        :param label: 样本的真实标签，shape=[batch_size] 的 tensor
        :param num_classes: 样本的类别总数，int
        :return: 分类错误的神经元的电压，与阈值电压之差的均方误差
        '''
        wrong_mask = ((v_max >= v_threshold).float() != F.one_hot(label, 10)).float()
        return torch.sum(torch.pow((v_max - v_threshold) * wrong_mask, 2)) / label.
↪shape[0]
```

下面我们直接运行代码。完整的源代码位于 `spikingjelly/event_driven/examples/tempotron_mnist.py`：

```
>>> import spikingjelly.event_driven.examples.tempotron_mnist as tempotron_mnist
>>> tempotron_mnist.main()
输入运行的设备，例如 “cpu” 或 “cuda:0”
 input device, e.g., "cpu" or "cuda:0": cuda:5
输入保存 MNIST 数据集的位置，例如 “./”
 input root directory for saving MNIST dataset, e.g., "./": ./mnist
输入 batch_size，例如 “64”
 input batch_size, e.g., "64": 64
输入学习率，例如 “1e-3”
 input learning rate, e.g., "1e-3": 1e-3
输入仿真时长，例如 “100”
 input simulating steps, e.g., "100": 10
```

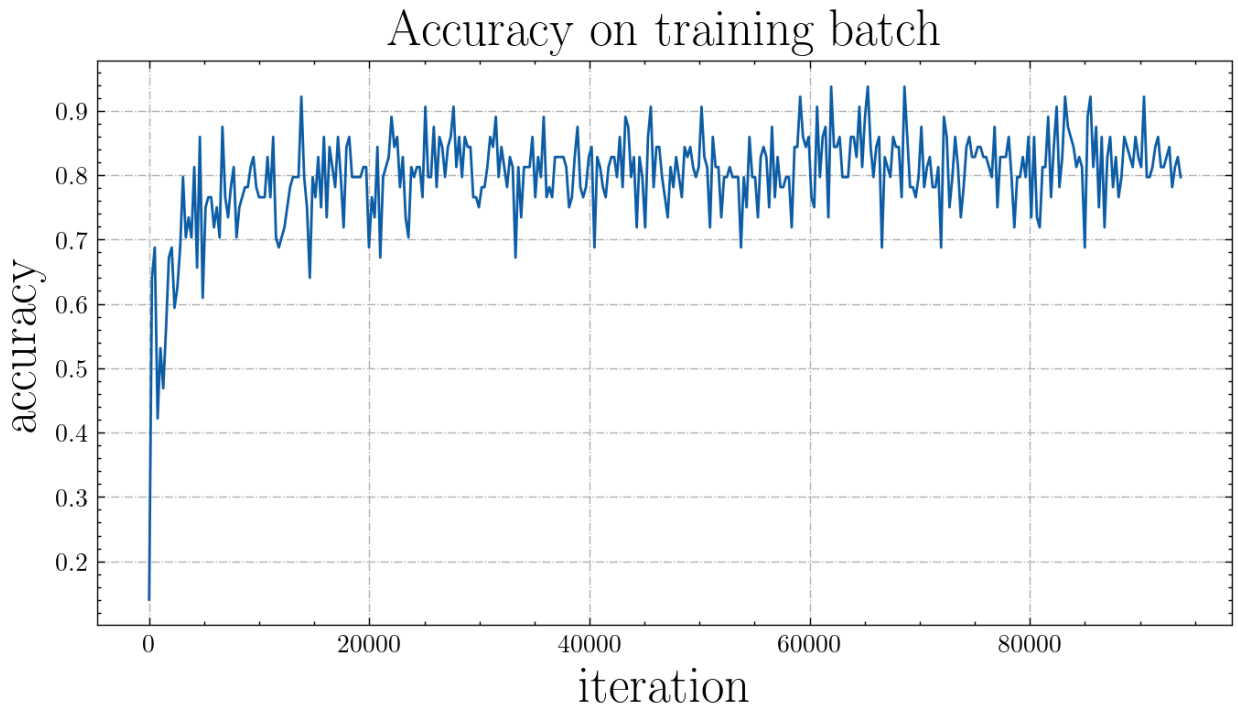
(下页继续)

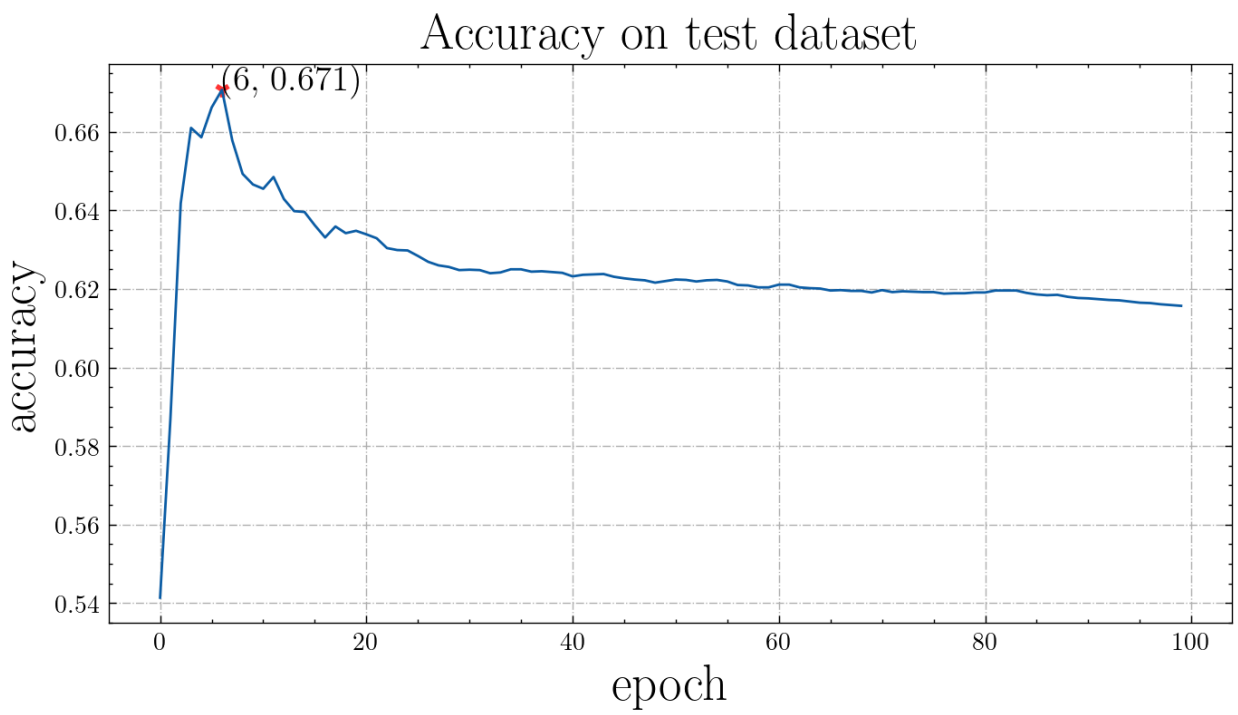
(续上页)

```
输入训练轮数，即遍历训练集的次数，例如“100”
 input training epochs, e.g., "100": 100
输入使用高斯调谐曲线编码每个像素点使用的神经元数量，例如“16”
 input neuron number for encoding a piexl in GaussianTuning encoder, e.g., "16": 16
输入保存 tensorboard 日志文件的位置，例如“./”
 input root directory for saving tensorboard logs, e.g., "./": ./logs
```

查看训练结果

在 Tesla K80 上训练 10 个 epoch，大约需要 32 分钟。训练时每个 batch 的正确率、测试集正确率的变化情况如下：





测试集的正确率 67.1% 左右，可以看出 Tempotron 确实实现了类似 ANN 中感知器的功能，具有一定的分类能力。但是与主流的多层 SNN 相比，性能较差。随着训练的进行，测试集正确率不断下降，过拟合比较严重。

1.3 时间驱动：神经元

本教程作者：fangwei123456

本节教程主要关注 `spikingjelly.clock_driven.neuron`，介绍脉冲神经元，和时间驱动的仿真方法。

1.3.1 脉冲神经元模型

在 `spikingjelly` 中，我们约定，只能输出脉冲，即 0 或 1 的神经元，都可以称之为“脉冲神经元”。使用脉冲神经元的网络，进而也可以称之为脉冲神经元网络（Spiking Neural Networks）。`spikingjelly.clock_driven.neuron` 中定义了各种常见的脉冲神经元模型，我们以 `spikingjelly.clock_driven.neuron.LIFNode` 为例来介绍脉冲神经元。

首先导入相关的模块：

```
import torch
import torch.nn as nn
import numpy as np
from spikingjelly.clock_driven import neuron
from spikingjelly import visualizing
from matplotlib import pyplot as plt
```

新建一个 LIF 神经元层：

```
lif = neuron.LIFNode()
```

LIF 神经元层有一些构造参数，在 API 文档中对这些参数有详细的解释：

- **tau** –膜电位时间常数。tau 对于这一层的所有神经元都是共享的
- **v_threshold** –神经元的阈值电压
- **v_reset** –神经元的重置电压。如果不为 None，当神经元释放脉冲后，电压会被重置为 v_reset；如果设置为 None，则电压会被减去 v_threshold
- **surrogate_function** –反向传播时用来计算脉冲函数梯度的替代函数
- **monitor_state** –是否设置监视器来保存神经元的电压和释放的脉冲。若为 True，则 self.monitor 是一个字典，键包括 h, v s，分别记录充电后的电压、释放脉冲后的电压、释放的脉冲。对应的值是一个链表。为了节省显存（内存），列表中存入的是原始变量转换为 numpy 数组后的值。还需要注意，self.reset() 函数会清空这些链表

其中 surrogate_function 参数，在前向传播时的行为与阶跃函数完全相同；我们暂时不会用到反向传播，因此可以先不关心反向传播。

你可能会好奇这一层神经元的数量是多少。对于 spikingjelly.clock_driven.neuron 中的绝大多数神经元层，神经元的数量是在初始化或调用 reset() 函数重新初始化后，根据第一次接收的输入的 shape 自动决定的。

与 RNN 中的神经元非常类似，脉冲神经元也是有状态的，或者说是有记忆。脉冲神经元的状态变量，一般是它的膜电位 V_t。因此，spikingjelly.clock_driven.neuron 中的神经元，都有成员变量 v。可以打印出刚才新建的 LIF 神经元层的膜电位：

```
print(lif.v)
# 0.0
```

可以发现，现在的 v 是 0.0，因为我们还没有给与它任何输入。我们给与几个不同的输入，观察神经元的电压的 shape，它与神经元的数量是一致的：

```
x = torch.rand(size=[2, 3])
lif(x)
print('x.shape', x.shape, 'lif.v.shape', lif.v.shape)
# x.shape torch.Size([2, 3]) lif.v.shape torch.Size([2, 3])
lif.reset()

x = torch.rand(size=[4, 5, 6])
lif(x)
print('x.shape', x.shape, 'lif.v.shape', lif.v.shape)
# x.shape torch.Size([4, 5, 6]) lif.v.shape torch.Size([4, 5, 6])
lif.reset()
```

那么 V_t 和输入 X_t 的关系是什么样的？在脉冲神经元中，不仅取决于当前时刻的输入 X_t，还取决于它在上一个时刻末的膜电位 V_{t-1}。

通常使用阈下（指的是膜电位不超过阈值电压 V_{threshold} 时）充电微分方程 $\frac{\mathrm{d}V(t)}{\mathrm{d}t} = f(V(t), X(t))$ 描述连续时间的脉冲神经元的充电过程，例如对于 LIF 神经元，充电方程为：

$$\tau_m \frac{\mathrm{d}V(t)}{\mathrm{d}t} = -(V(t) - V_{reset}) + X(t)$$

其中 τ_m 是膜电位时间常数，V_{reset} 是重置电压。对于这样的微分方程，由于 $X(t)$ 并不是常量，因此难以求出显示的解析解。

`spikingjelly.clock_driven.neuron` 中的神经元，使用离散的差分方程来近似连续的微分方程。在差分方程的视角下，LIF 神经元的充电方程为：

$$\tau_m(V_t - V_{t-1}) = -(V_{t-1} - V_{reset}) + X_t$$

因此可以得到 V_t 的表达式为

$$V_t = f(V_{t-1}, X_t) = V_{t-1} + \frac{1}{\tau_m}(-(V_{t-1} - V_{reset}) + X_t)$$

可以在 `LIFNode` 的 `neuronal_charge()` 中找到对应的代码：

```
def neuronal_charge(self, dv: torch.Tensor):
    if self.v_reset is None:
        self.v += (dv - self.v) / self.tau
    else:
        self.v += (dv - (self.v - self.v_reset)) / self.tau
```

不同的神经元，充电方程不尽相同。但膜电位超过阈值电压后，释放脉冲，以及释放脉冲后，膜电位的重置都是相同的。因此它们全部继承自 `BaseNode`，共享相同的放电、重置方程。可以在 `BaseNode` 的 `neuronal_fire()` 中找到释放脉冲的代码：

```
def neuronal_fire(self):
    self.spike = self.surrogate_function(self.v - self.v_threshold)
```

`surrogate_function()` 在前向传播时是阶跃函数，只要输入大于或等于 0，就会返回 1，否则会返回 0。我们将这种元素仅为 0 或 1 的 `tensor` 视为脉冲。

释放脉冲消耗了神经元之前积累的电荷，因此膜电位会有一个瞬间的降低，即膜电位的重置。在 SNN 中，对膜电位重置的实现，有 2 种方式：

1. Hard 方式：释放脉冲后，膜电位直接被设置成重置电压：$V = V_{reset}$
2. Soft 方式：释放脉冲后，膜电位减去阈值电压：$V = V - V_{threshold}$

可以发现，对于使用 Soft 方式的神经元，并不需要重置电压 V_{reset} 这个变量。`spikingjelly.clock_driven.neuron` 中的神经元，在构造函数的参数之一 `v_reset`，默认为 `1.0`，表示神经元使用 Hard 方式；若设置为 `None`，则会使用 Soft 方式。在 `BaseNode` 的 `neuronal_reset()` 中找到膜电位重置的代码：

```
def neuronal_reset(self):
    if self.detach_reset:
        spike = self.spike.detach()
    else:
        spike = self.spike

    if self.v_reset is None:
        self.v = self.v - spike * self.v_threshold
    else:
        self.v = (1 - spike) * self.v + spike * self.v_reset
```

1.3.2 描述离散脉冲神经元的三个方程

至此，我们可以用充电、放电、重置，这 3 个离散方程来描述任意的离散脉冲神经元。充电、放电方程为：

$$
\begin{aligned}
H_t &= f(V_{t-1}, X_t) \\
S_t &= g(H_t - V_{threshold}) = \Theta(H_t - V_{threshold})
\end{aligned}
$$

其中 $\Theta(x)$ 即为构造函数参数中的 `surrogate_function()`，是一个阶跃函数：

$$
\Theta(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}
$$

Hard 方式重置方程为：

$$
V_t = H_t \cdot (1 - S_t) + V_{reset} \cdot S_t
$$

Soft 方式重置方程为：

$$
V_t = H_t - V_{threshold} \cdot S_t
$$

其中 V_t 是神经元的膜电位；X_t 是外源输入，例如电压增量；为了避免混淆，我们使用 H_t 表示神经元充电后、释放脉冲前的膜电位；V_t 是神经元释放脉冲后的膜电位；$f(V_{t-1}, X_t)$ 是神经元的状态更新方程，不同的神经元，区别就在于更新方程不同。

1.3.3 时间驱动的仿真方式

`spikingjelly.clock_driven` 使用时间驱动的方式，对 SNN 逐步进行仿真。

接下来，我们将逐步给与神经元输入，并查看它的膜电位和输出脉冲。为了记录数据，只需要将神经元层的监视器 `monitor` 打开：

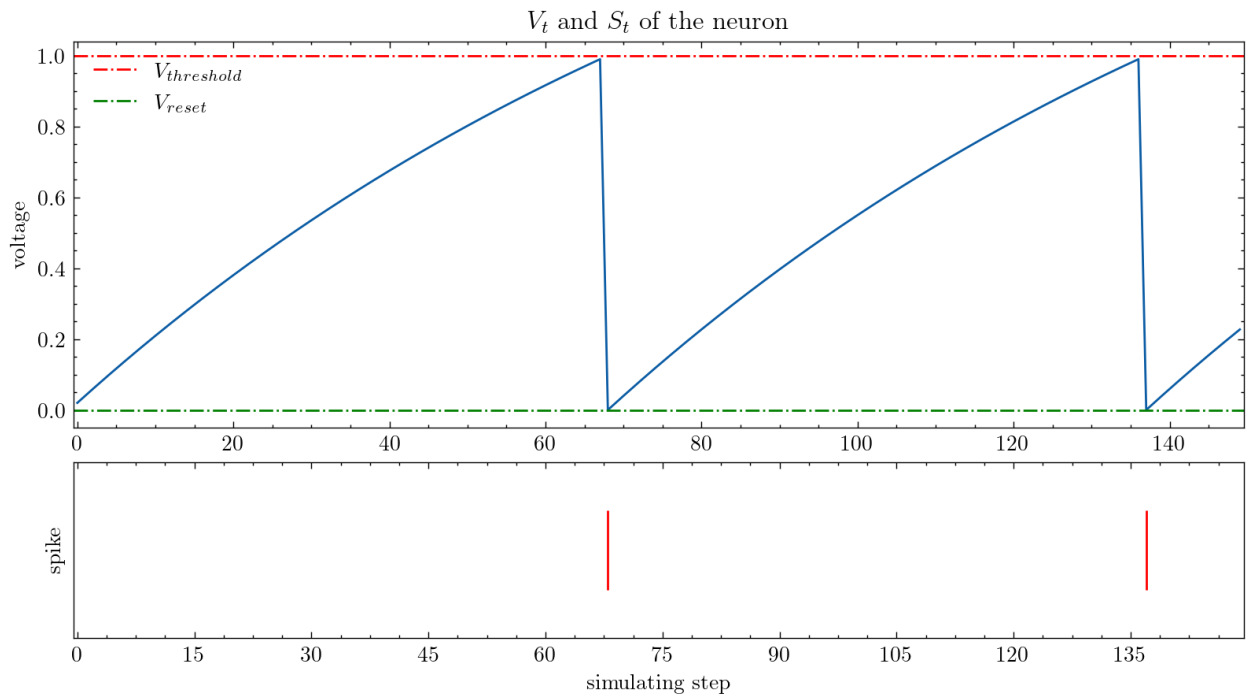
```
lif.set_monitor(True)
```

在打开监视器后，神经元层在运行时，会在字典 `self.monitor` 中自动记录运行过程中的充电后的膜电位 `self.monitor['h']`，释放的脉冲 `self.monitor['s']`，和放电后的膜电位 `self.monitor['v']`。

现在让我们给与 LIF 神经元层持续的输入，并画出其放电后的膜电位和输出脉冲：

```
x = torch.Tensor([2.0])
T = 150
for t in range(T):
    lif(x)
visualizing.plot_one_neuron_v_s(lif.monitor['v'], lif.monitor['s'], v_threshold=lif.v_
↪threshold, v_reset=lif.v_reset, dpi=200)
plt.show()
```

我们给与的输入 `shape=[1]`，因此这个 LIF 神经元层只有 1 个神经元。它的膜电位和输出脉冲随着时间变化情况如下：



下面我们将神经元层重置，并给与 `shape=[32]` 的输入，查看这 32 个神经元的膜电位和输出脉冲：

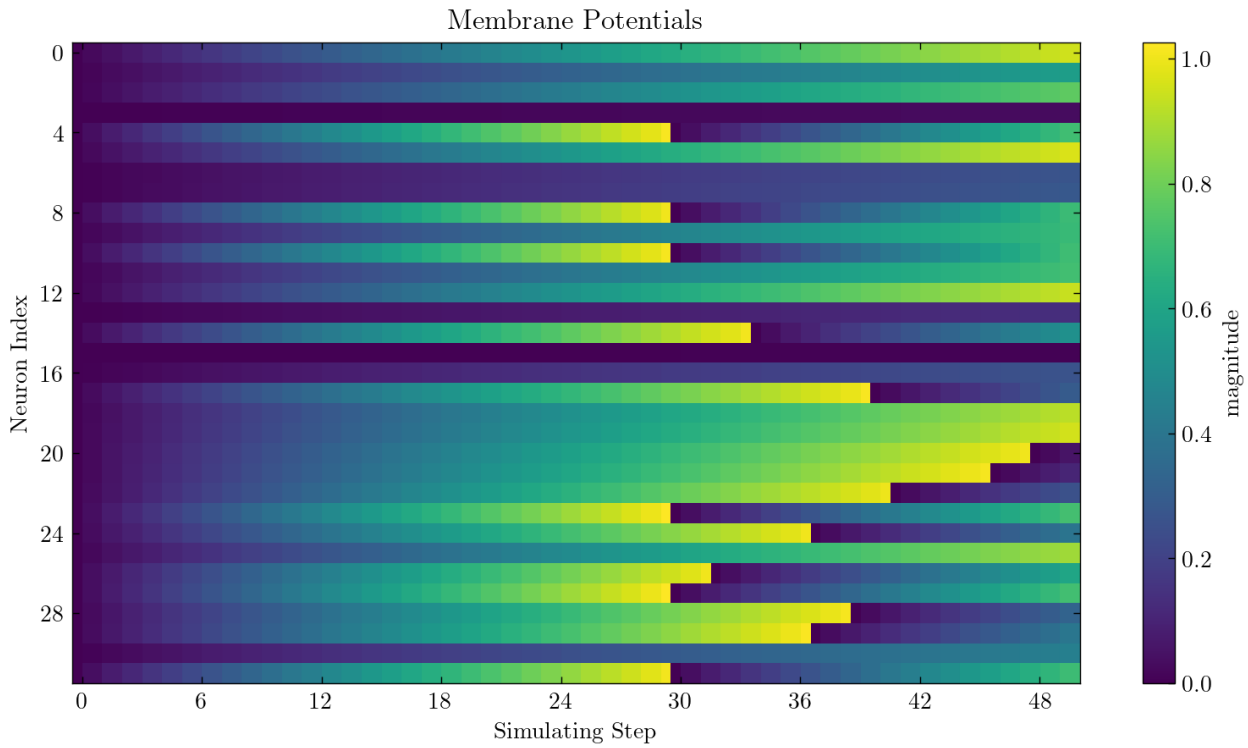
```
lif.reset()
x = torch.rand(size=[32]) * 4
T = 50
for t in range(T):
    lif(x)

visualizing.plot_2d_heatmap(array=np.asarray(lif.monitor['v']).T, title='Membrane␣
↪Potentials', xlabel='Simulating Step',
                                    ylabel='Neuron Index', int_x_ticks=True, x_max=T,␣
↪dpi=200)
```

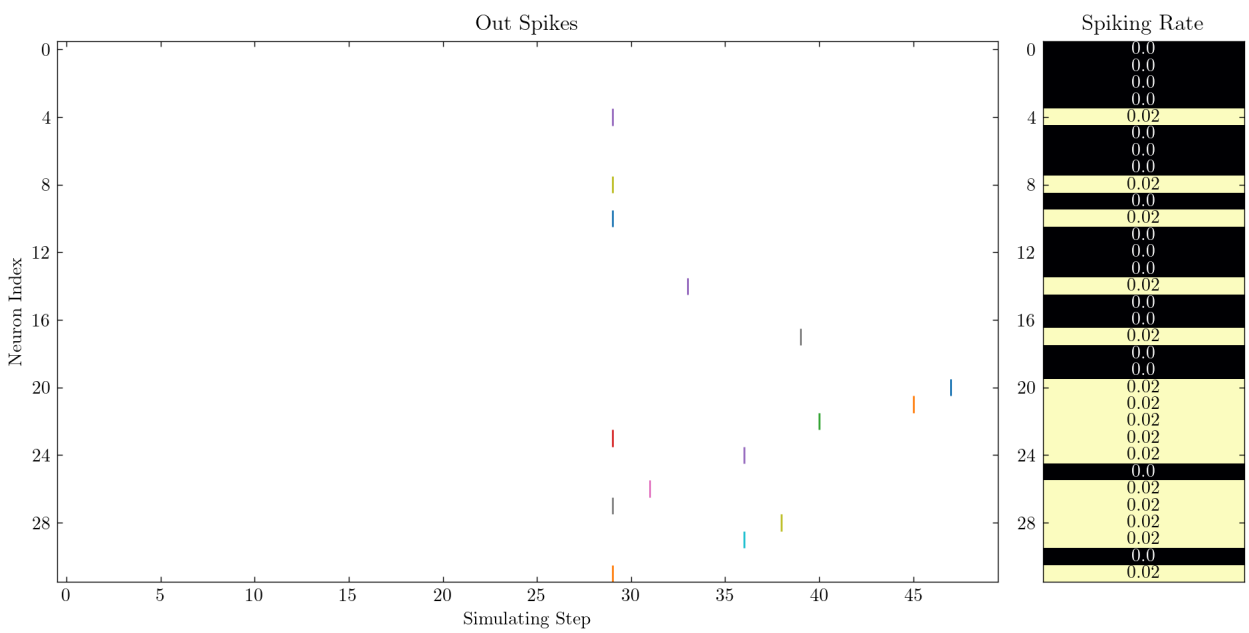
(下页继续)

(续上页)

```
visualizing.plot_1d_spikes(spikes=np.asarray(lif.monitor['s']).T, title='Membrane␣
↪Potentials', xlabel='Simulating Step',
                                    ylabel='Neuron Index', dpi=200)
plt.show()
```

结果如下：





1.4 时间驱动：编码器

本教程作者：Grasshlw, Yanqi-Chen

本节教程主要关注 spikingjelly.clock_driven.encoding ，介绍编码器。

1.4.1 编码器基类

在 spikingjelly.clock_driven 中，所定义的编码器都继承自编码器基类 BaseEncoder ，该编码器继承 torch.nn.Module ，定义三个方法，其一 forward 将输入数据 x 编码为脉冲；其二 step 针对多数编码器，x 被编码成一定长度的脉冲序列，需进行多步输出，则用 step 获取每一步的脉冲数据；其三 reset 将编码器的状态变量设置为初始状态。

```
class BaseEncoder(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        raise NotImplementedError

    def step(self):
        raise NotImplementedError

    def reset(self):
        pass
```

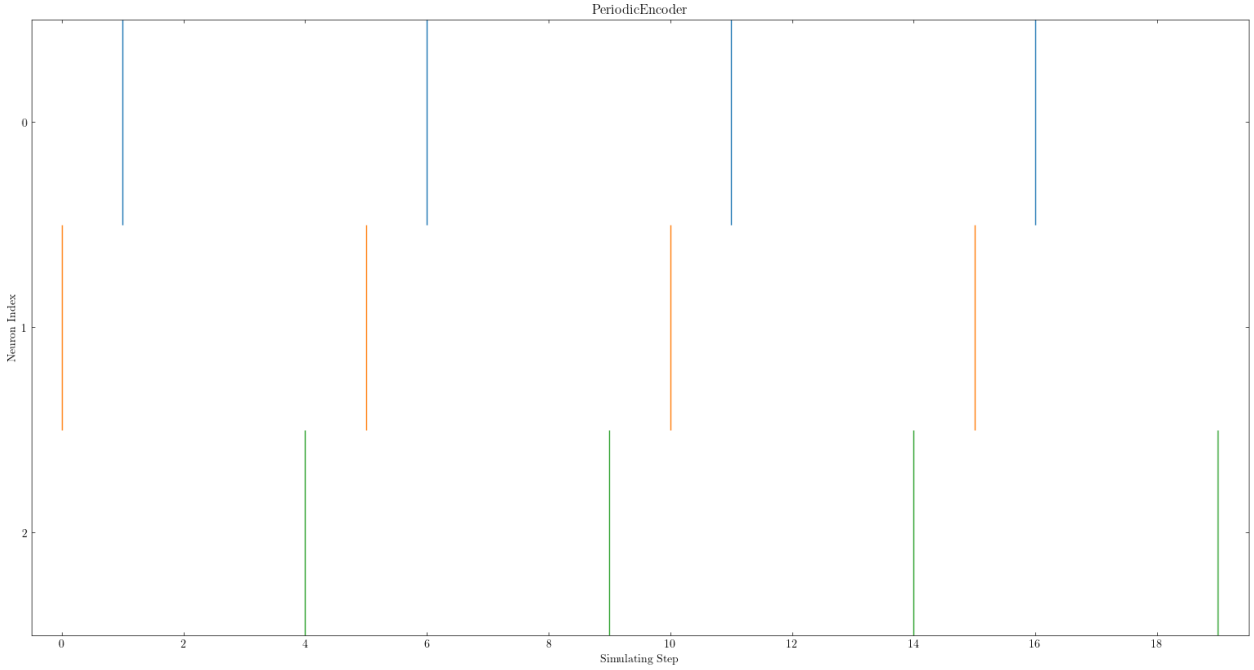
1.4.2 周期编码器

周期编码器是周期性输出给定的脉冲序列的编码器。与输入的数据无关，类 PeriodicEncoder 在初始化时已设定好要输出的脉冲序列 out_spike ，并可在使用过程中通过方法 set_out_spike 重新设定。

```
class PeriodicEncoder(BaseEncoder):
    def __init__(self, out_spike):
        super().__init__()
        assert out_spike.dtype == torch.bool
        self.out_spike = out_spike
        self.T = out_spike.shape[0]
        self.index = 0
```

示例：给定 3 个神经元，时间步长为 5 的脉冲序列，分别为 01000 、10000 、00001 。初始化周期编码器，输出 20 个时间步长的仿真脉冲数据。

```
import torch
import matplotlib
import matplotlib.pyplot as plt
from spikingjelly.clock_driven import encoding
from spikingjelly import visualizing

# 给定脉冲序列
set_spike = torch.full((3, 5), 0, dtype=torch.bool)
set_spike[0, 1] = 1
set_spike[1, 0] = 1
set_spike[2, 4] = 1

pe = encoding.PeriodicEncoder(set_spike.transpose(0, 1))

# 输出周期性编码器的编码结果
out_spike = torch.full((3, 20), 0, dtype=torch.bool)
for t in range(out_spike.shape[1]):
    out_spike[:, t] = pe.step()

plt.style.use(['science', 'muted'])
visualizing.plot_1d_spikes(out_spike.float().numpy(), 'PeriodicEncoder', 'Simulating␣
↪Step', 'Neuron Index',
                           plot_firing_rate=False)
plt.show()
```



1.4.3 延迟编码器

延迟编码器是根据输入数据 `x`，延迟发放脉冲的编码器。当刺激强度越大，发放时间就越早，且存在最大脉冲发放时间。因此对于每一个输入数据 `x`，都能得到一段时间步长为最大脉冲发放时间的脉冲序列，每段序列有且仅有一个脉冲发放。

脉冲发放时间 t_i 与刺激强度 x_i 满足以下二式：当编码类型为线性时（`function_type='linear'`）

$$t_i = (t_{max} - 1) * (1 - x_i)$$

当编码类型为对数时（`function_type='log'`）

$$t_i = (t_{max} - 1) - ln(\alpha * x_i + 1)$$

其中，t_{max} 为最大脉冲发放时间，x_i 需归一化到 $[0, 1]$。

考虑第二个式子，α 需满足：

$$(t_{max} - 1) - ln(\alpha * 1 + 1) = 0$$

这会导致该编码器很可能发生溢出，因为

$$\alpha = e^{t_{max}-1} - 1$$

α 会随着 t_{max} 增大而指数增长，最终造成溢出。

示例：随机生成 6 个 `x`，分别为 6 个神经元的刺激强度，并设定最大脉冲发放时间为 20，对以上输入数据进行编码。

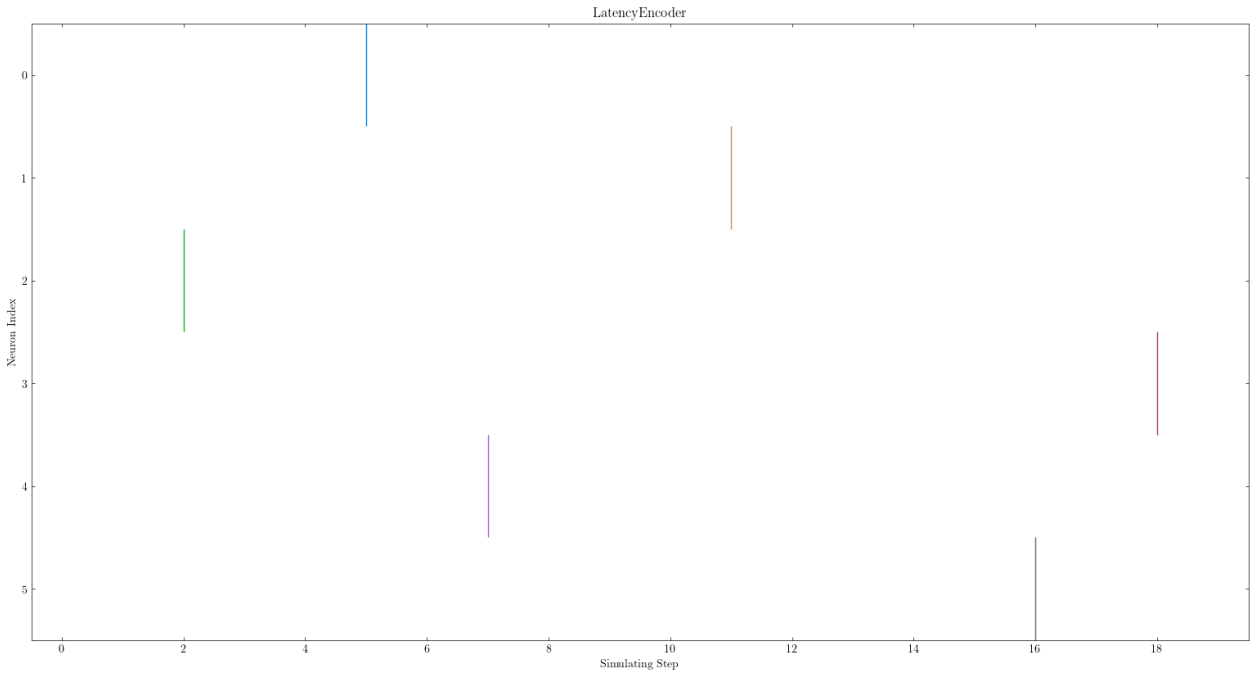
```
import torch
import matplotlib
import matplotlib.pyplot as plt
from spikingjelly.clock_driven import encoding
from spikingjelly import visualizing

# 随机生成 6 个神经元的刺激强度，设定最大脉冲时间为 20
x = torch.rand(6)
max_spike_time = 20

# 将输入数据编码为脉冲序列
le = encoding.LatencyEncoder(max_spike_time)
le(x)

# 输出延迟编码器的编码结果
out_spike = torch.full((6, 20), 0, dtype=torch.bool)
for t in range(max_spike_time):
    out_spike[:, t] = le.step()
```

(下页继续)

(续上页)

```
print(x)
plt.style.use(['science', 'muted'])
visualizing.plot_1d_spikes(out_spike.float().numpy(), 'LatencyEncoder', 'Simulating␣
↪Step', 'Neuron Index',
                           plot_firing_rate=False)
plt.show()
```

当随机生成的 6 个刺激强度分别为 0.6650 、0.3704 、0.8485 、0.0247 、0.5589 和 0.1030 时，得到的脉冲序列如下：



1.4.4 泊松编码器

泊松编码器将输入数据 x 编码为发放次数分布符合泊松过程的脉冲序列。泊松过程又被称为泊松流，当一个脉冲流满足独立增量性、增量平稳性和普通性时，这样的脉冲流就是一个泊松流。更具体地说，在整个脉冲流中，互不相交的区间里出现脉冲的个数是相互独立的，且在任意一个区间中，出现脉冲的个数与区间的起点无关，与区间的长度有关。因此，为了实现泊松编码，我们令一个时间步长的脉冲发放概率 $p = x$, 其中 x 需归一化到 [0,1]。

示例：输入图像为 lena512.bmp ，仿真 20 个时间步长，得到 20 个脉冲矩阵。

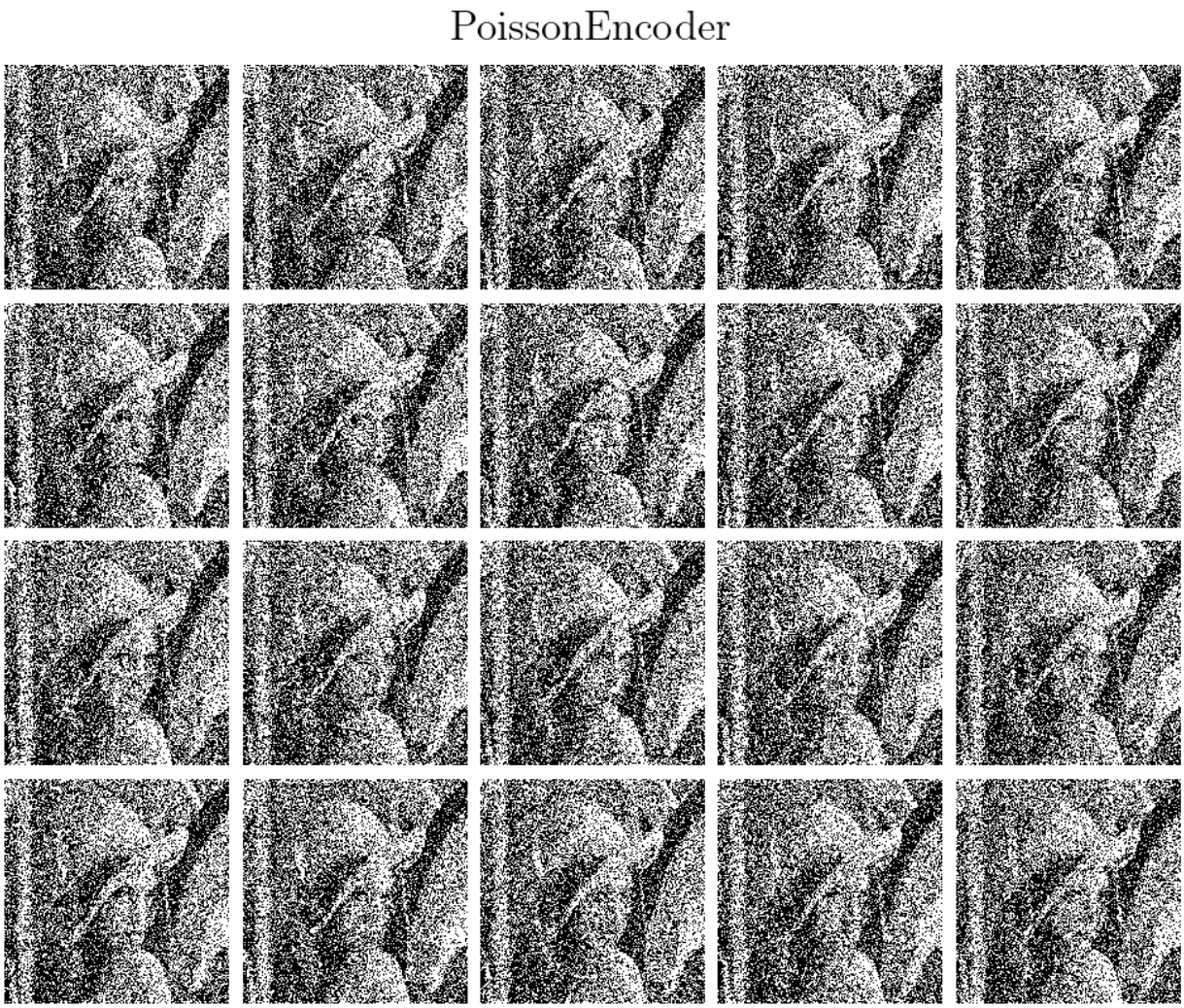
```
import torch
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
```

(下页继续)

(续上页)

```
from PIL import Image
from spikingjelly.clock_driven import encoding
from spikingjelly import visualizing

# 读入 lena 图像
lena_img = np.array(Image.open('lena512.bmp')) / 255
x = torch.from_numpy(lena_img)

pe = encoding.PoissonEncoder()

# 仿真 20 个时间步长，将图像编码为脉冲矩阵并输出
w, h = x.shape
out_spike = torch.full((20, w, h), 0, dtype=torch.bool)
T = 20
for t in range(T):
    out_spike[t] = pe(x)

plt.figure()
plt.style.use(['science', 'muted'])
plt.imshow(x, cmap='gray')
plt.axis('off')

visualizing.plot_2d_spiking_feature_map(out_spike.float().numpy(), 4, 5, 30,
↪'PoissonEncoder')
plt.axis('off')
```

lena 原灰度图和编码后 20 个脉冲矩阵如下：



对比原灰度图和编码后的脉冲矩阵，可发现脉冲矩阵很接近原灰度图的轮廓，可见泊松编码器性能的优越性。

同样对 lena 灰度图进行编码，仿真 512 个时间步长，将每一步得到的脉冲矩阵叠加，得到第 1、128、256、384、512 步叠加得到的结果并画图：

```
# 仿真 512 个时间不长，将编码的脉冲矩阵逐次叠加，得到第 1、128、256、384、512 次叠加的结果并输出
superposition = torch.full((w, h), 0, dtype=torch.float)
superposition_ = torch.full((5, w, h), 0, dtype=torch.float)
T = 512
for t in range(T):
    superposition += pe(x).float()
    if t == 0 or t == 127 or t == 255 or t == 387 or t == 511:
        superposition_[int((t + 1) / 128)] = superposition

# 归一化
for i in range(5):
    min_ = superposition_[i].min()
```

(下页继续)

(续上页)

```
    max_ = superposition_[i].max()
    superposition_[i] = (superposition_[i] - min_) / (max_ - min_)

# 画图
visualizing.plot_2d_spiking_feature_map(superposition_.numpy(), 1, 5, 30,
↪'PoissonEncoder')
plt.axis('off')

plt.show()
```

叠加后的图像如下：

可见当仿真足够的步长，泊松编码器得到的脉冲叠加后几乎可以重构出原始图像。

1.4.5 高斯调谐曲线编码器

对于有 `M` 个特征的输入数据，高斯调谐曲线编码器使用 `tuning_curve_num` 个神经元去编码输入数据的每一个特征，将每个特征编码为这 `tuning_curve_num` 个神经元的脉冲发放时刻，因此可认为编码器有 `M` × `tuning_curve_num` 个神经元在工作。

对于第 i 个特征 X^i，取值范围为 $X^i_{min} <= X^i <= X^i_{max}$。根据特征最大和最小值可计算出 `tuning_curve_num` 条高斯曲线 G^i_j 的均值和方差：

$$\mu^i_j = x^i_{min} + \frac{2j-3}{2}\frac{x^i_{max}-x^i_{min}}{m-2}\sigma^i_j = \frac{1}{\beta}\frac{x^i_{max}-x^i_{min}}{m-2}$$

其中 β 通常取值 1.5。对于同一个特征，所有高斯曲线形状完全相同，对称轴位置不同。

生成高斯曲线后，则计算每个输入对应的高斯函数值，并将这些函数值线性转换为 `[0, max_spike_time - 1]` 之间的脉冲发放时间。此外，对于最后时刻发放的脉冲，被认为是没有脉冲发放。

根据以上步骤，完成对输入数据的编码。

1.4.6 间隔编码器

间隔编码器是每隔 `T` 个时间步长发放一次脉冲的编码器。该编码器较为简单，此处不再详述。

1.4.7 带权相位编码器

一种基于二进制表示的编码方法。

将输入数据按照二进制各位展开，从高位到低位遍历输入进行脉冲编码。相比于频率编码，每一位携带的信息量更多。编码相位数为 K 时，可以对于处于区间 $[0, 1-2^{-K}]$ 的数进行编码。以下为原始论文[1] 中 $K=8$ 的示例：

Phase (K=8)	1	2	3	4	5	6	7	8
Spike weight $\omega(t)$	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
192/256	1	1	0	0	0	0	0	0
1/256	0	0	0	0	0	0	0	1
128/256	1	0	0	0	0	0	0	0
255/256	1	1	1	1	1	1	1	1

1.5 时间驱动：使用单层全连接 SNN 识别 MNIST

本教程作者：Yanqi-Chen

本节教程将介绍如何使用编码器与替代梯度方法训练一个最简单的 MNIST 分类网络。

1.5.1 从头搭建一个简单的 SNN 网络

在 PyTorch 中搭建神经网络时，我们可以简单地使用 `nn.Sequential` 将多个网络层堆叠得到一个前馈网络，输入数据将依序流经各个网络层得到输出。

MNIST 数据集包含若干尺寸为 28×28 的 8 位灰度图像，总共有 0~9 共 10 个类别。以 MNIST 的分类为例，一个简单的单层 ANN 网络如下：

```
net = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 10, bias=False),
    nn.Softmax()
    )
```

我们也可以用完全类似结构的 SNN 来进行分类任务。就这个网络而言，只需要先去掉所有的激活函数，再将神经元添加到原来激活函数的位置，这里我们选择的是 LIF 神经元：

[1] Kim J, Kim H, Huh S, et al. Deep neural networks with weighted spikes[J]. Neurocomputing, 2018, 311: 373-386.

```
net = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 10, bias=False),
    neuron.LIFNode(tau=tau)
    )
```

其中膜电位衰减常数 τ 需要通过参数 `tau` 设置。

1.5.2 训练 SNN 网络

首先指定好训练参数以及若干其他配置

```
device = input('输入运行的设备，例如 “cpu” 或 “cuda:0” \n input device, e.g., "cpu" or
↪"cuda:0": ')
dataset_dir = input('输入保存 MNIST 数据集的位置，例如 “./” \n input root directory for␣
↪saving MNIST dataset, e.g., "./": ')
batch_size = int(input('输入 batch_size，例如 “64” \n input batch_size, e.g., "64": '))
learning_rate = float(input('输入学习率，例如 “1e-3” \n input learning rate, e.g., "1e-3
↪": '))
T = int(input('输入仿真时长，例如 “100” \n input simulating steps, e.g., "100": '))
tau = float(input('输入 LIF 神经元的时间常数 tau，例如 “100.0” \n input membrane time␣
↪constant, tau, for LIF neurons, e.g., "100.0": '))
train_epoch = int(input('输入训练轮数，即遍历训练集的次数，例如 “100” \n input training␣
↪epochs, e.g., "100": '))
log_dir = input('输入保存 tensorboard 日志文件的位置，例如 “./” \n input root directory for␣
↪saving tensorboard logs, e.g., "./": ')
```

优化器使用 Adam，以及使用泊松编码器，在每次输入图片时进行脉冲编码

```
# 使用 Adam 优化器
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
# 使用泊松编码器
encoder = encoding.PoissonEncoder()
```

训练代码的编写需要遵循以下三个要点：

1. 脉冲神经元的输出是二值的，而直接将单次运行的结果用于分类极易受到干扰。因此一般认为脉冲网络的输出是输出层一段时间内的**发放频率**（或称发放率），发放率的高低表示该类别的响应大小。因此网络需要运行一段时间，即使用 `T` 个时刻后的**平均发放率**作为分类依据。
2. 我们希望的理想结果是除了正确的神经元**以最高频率发放**，其他神经元**保持静默**。常常采用交叉熵损失或者 MSE 损失，这里我们使用实际效果更好的 MSE 损失。
3. 每次网络仿真结束后，需要**重置**网络状态

结合以上三点，得到训练循环的代码如下：

```
for img, label in train_data_loader:
    img = img.to(device)
    label = label.to(device)
    label_one_hot = F.one_hot(label, 10).float()

    optimizer.zero_grad()

    # 运行 T 个时长，out_spikes_counter 是 shape=[batch_size, 10] 的 tensor
    # 记录整个仿真时长内，输出层的 10 个神经元的脉冲发放次数
    for t in range(T):
        if t == 0:
            out_spikes_counter = net(encoder(img).float())
        else:
            out_spikes_counter += net(encoder(img).float())

    # out_spikes_counter / T 得到输出层 10 个神经元在仿真时长内的脉冲发放频率
    out_spikes_counter_frequency = out_spikes_counter / T

    # 损失函数为输出层神经元的脉冲发放频率，与真实类别的 MSE
    # 这样的损失函数会使，当类别 i 输入时，输出层中第 i 个神经元的脉冲发放频率趋近 1，而其他神经元的脉
冲发放频率趋近 0
    loss = F.mse_loss(out_spikes_counter_frequency, label_one_hot)
    loss.backward()
    optimizer.step()
    # 优化一次参数后，需要重置网络的状态，因为 SNN 的神经元是有“记忆”的
    functional.reset_net(net)
```

完整的代码位于 `clock_driven.examples.lif_fc_mnist.py`，在代码中我们还使用了 Tensorboard 来保存训练日志。可以直接在 Python 命令行运行它：

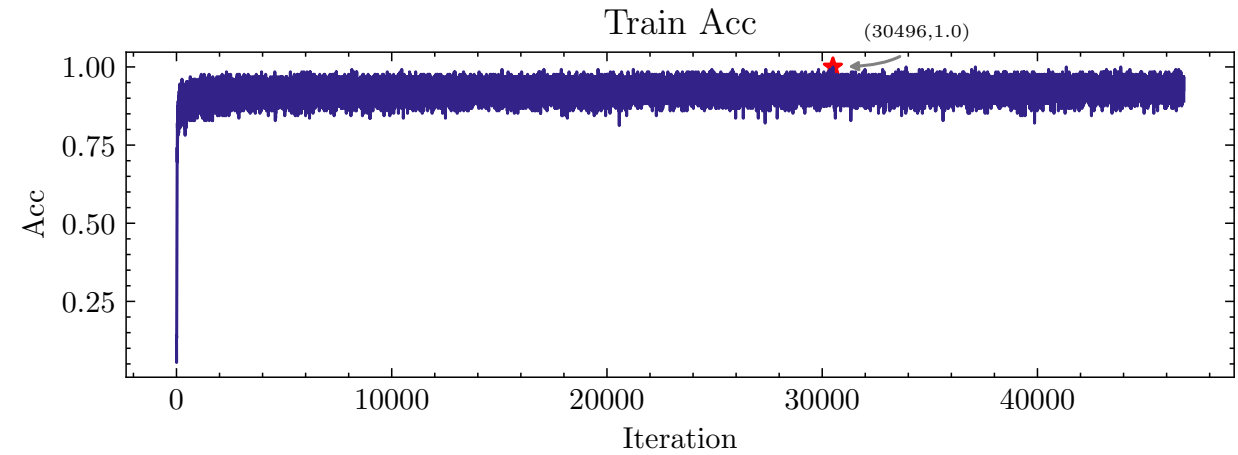
```
>>> import spikingjelly.clock_driven.examples.lif_fc_mnist as lif_fc_mnist
>>> lif_fc_mnist.main()
```

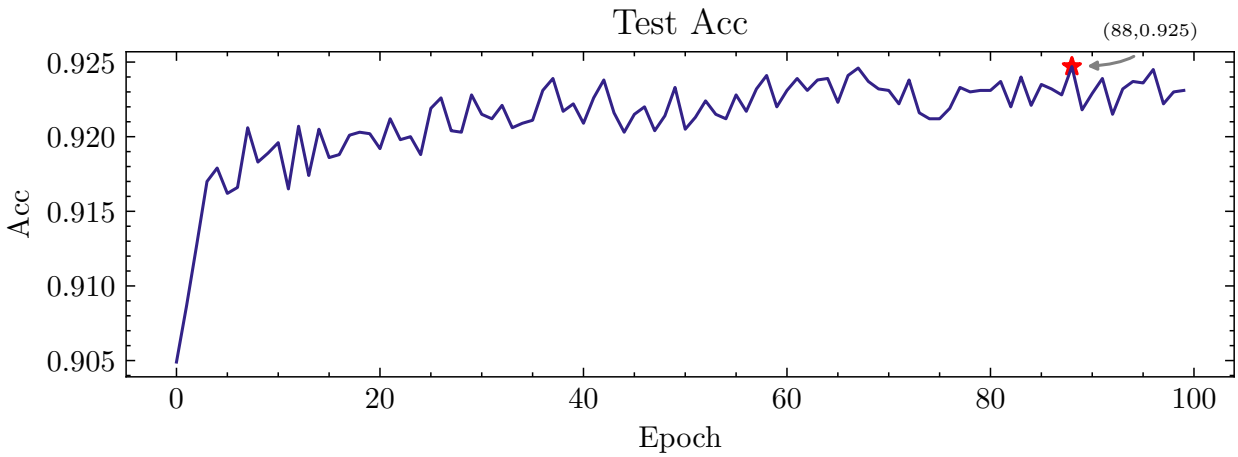
需要注意的是，训练这样的 SNN，所需显存数量与仿真时长 `T` 线性相关，更长的 `T` 相当于使用更小的仿真步长，训练更为“精细”，但训练效果不一定更好。`T` 太大时，SNN 在时间上展开后会变成一个非常深的网络，这将导致梯度的传递容易衰减或爆炸。

另外由于我们使用了泊松编码器，因此需要较大的 `T`。

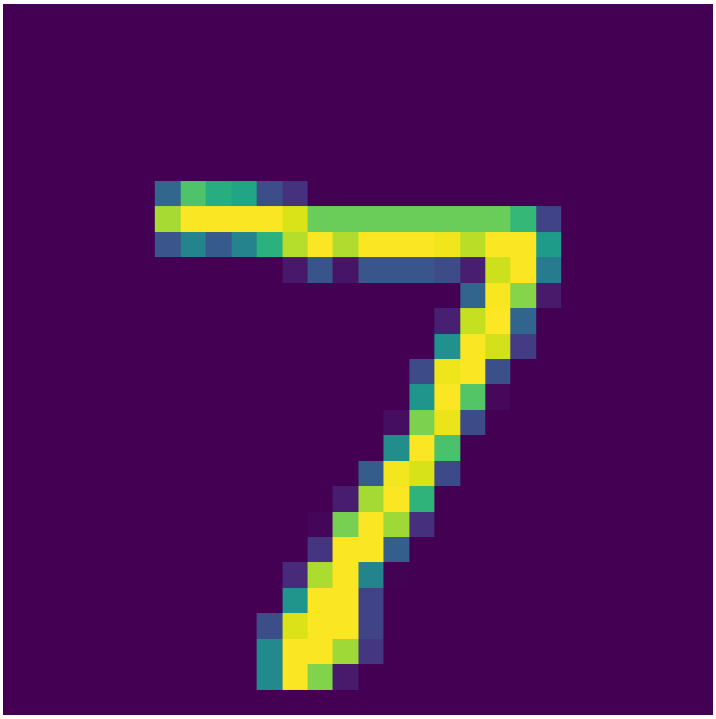
1.5.3 训练结果

取 `tau=2.0,T=100,batch_size=128,lr=1e-3`，训练 100 个 Epoch 后，将会输出四个 npy 文件。测试集上的最高正确率为 92.5%，通过 matplotlib 可视化得到的正确率曲线如下





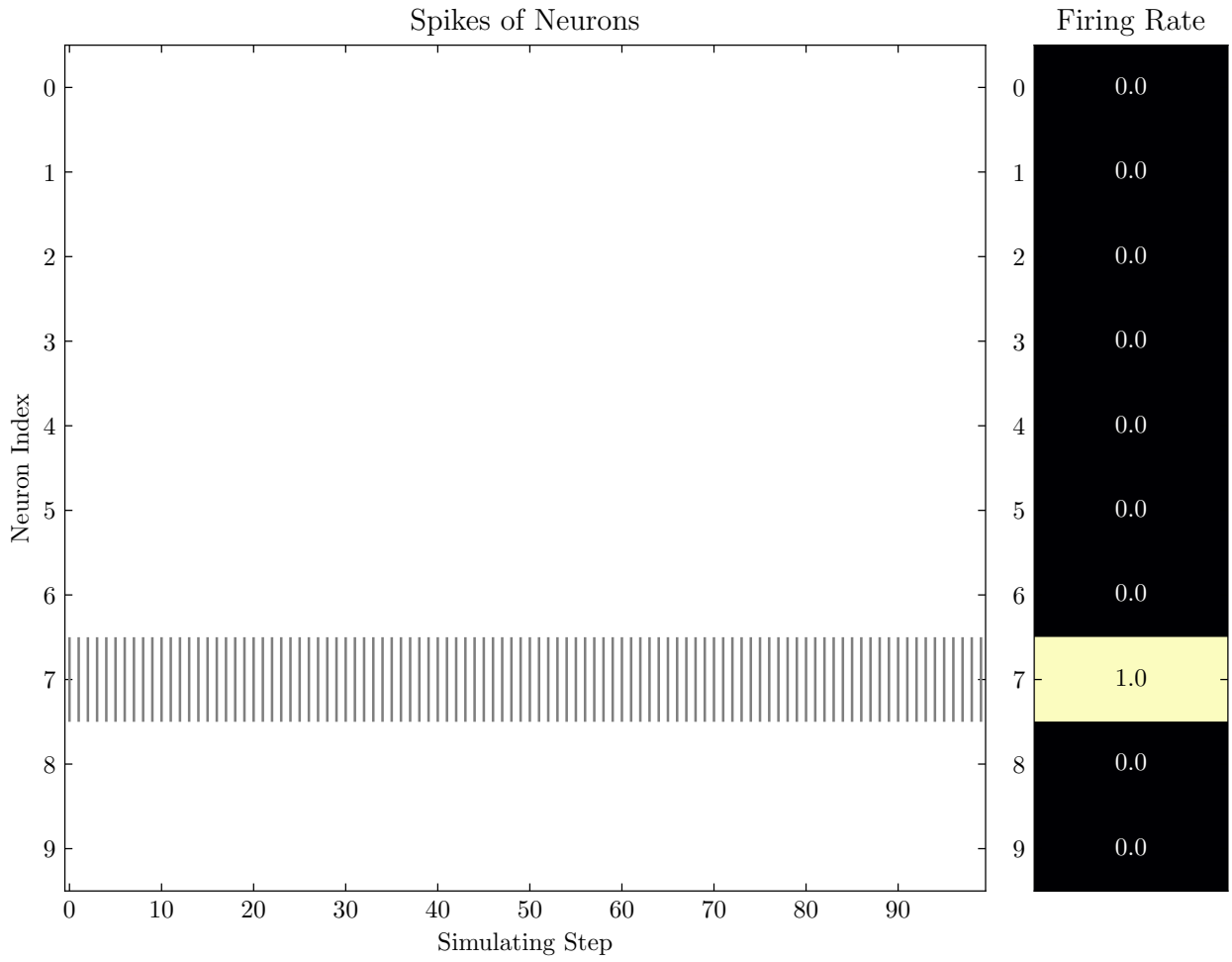
选取测试集中第一张图片：

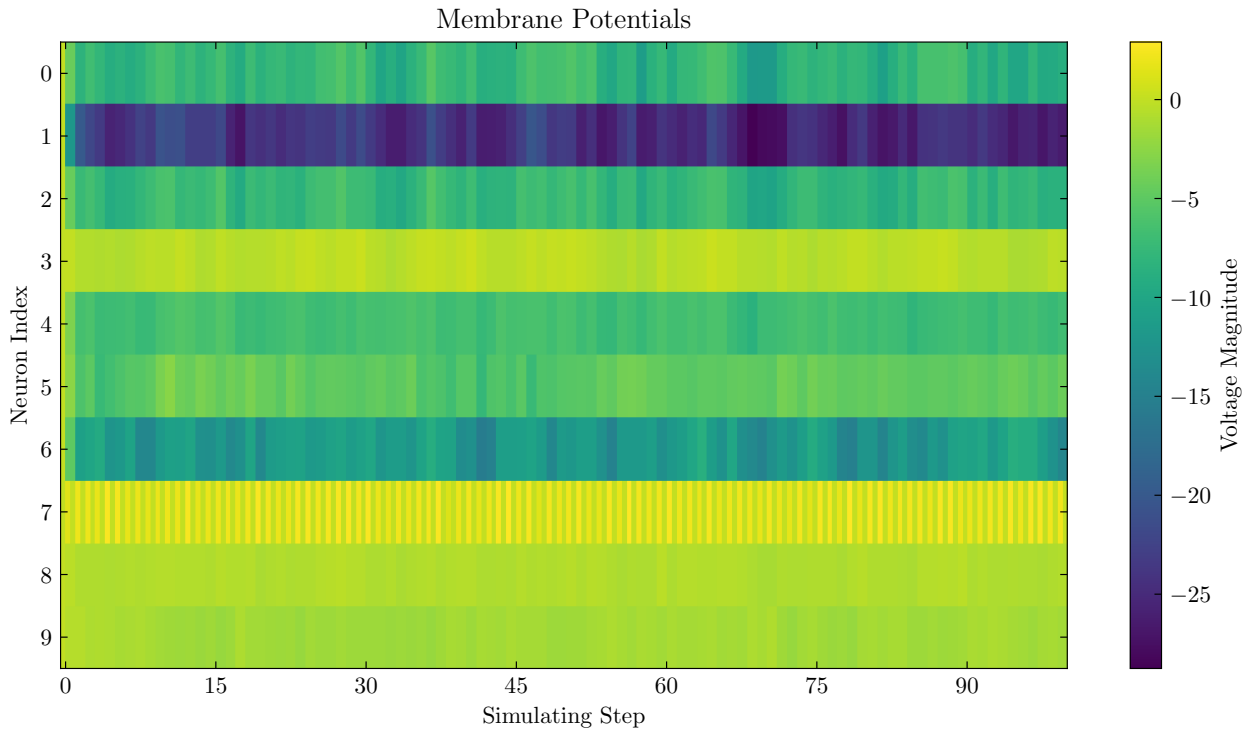


用训好的模型进行分类，得到分类结果

```
Firing rate: [[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]]
```

通过 `visualizing` 模块中的函数可视化得到输出层的电压以及脉冲如下图所示


Spikes of Neurons
Firing Rate
Neuron Index
Simulating Step
0
1
2
3
4
5
6
7
8
9
0
10
20
30
40
50
60
70
80
90
0.0
0.0
0.0
0.0
0.0
0.0
0.0
1.0
0.0
0.0



可以看到除了正确类别对应的神经元外，其它神经元均未发放任何脉冲。完整的训练代码可见 clock_driven/examples/lif_fc_mnist.py 。

1.6 时间驱动：使用卷积 SNN 识别 Fashion-MNIST

本教程作者：fangwei123456

在本节教程中，我们将搭建一个卷积脉冲神经网络，对 Fashion-MNIST 数据集进行分类。Fashion-MNIST 数据集，与 MNIST 数据集的格式相同，均为 `1 * 28 * 28` 的灰度图片。

1.6.1 网络结构

ANN 中常见的卷积神经网络，大多数是卷积 + 全连接层的形式，我们在 SNN 中也使用类似的结构。导入相关的模块，继承 `torch.nn.Module`，定义我们的网络：

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
from spikingjelly.clock_driven import neuron, functional, surrogate, layer
from torch.utils.tensorboard import SummaryWriter
import sys
if sys.platform != 'win32':
```

(下页继续)

(续上页)

```
    import readline
class Net(nn.Module):
    def __init__(self, tau, v_threshold=1.0, v_reset=0.0):
```

接下来，我们在 Net 的成员变量中添加卷积层和全连接层。SpikingJelly 的开发者们在实验中发现，对于不含时间信息、静态的图片数据，卷积层中的神经元用 IFNode 效果更好一些。我们添加 2 个卷积-BN-池化层：

```
self.conv = nn.Sequential(
        nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(128),
        neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
        nn.MaxPool2d(2, 2),  # 14 * 14

        nn.Conv2d(128, 128, kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(128),
        neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
        nn.MaxPool2d(2, 2)  # 7 * 7
    )
```

1 * 28 * 28 的输入经过这样的卷积层作用后，得到 128 * 7 * 7 的输出脉冲。

这样的卷积层，其实可以起到编码器的作用：在上一届教程，MNIST 识别的代码中，我们使用泊松编码器，将图片编码成脉冲。实际上我们完全可以直接将图片送入 SNN，在这种情况下，SNN 中的首层脉冲神经元层及其之前的层，可以看作是一个参数可学习的自编码器。例如我们刚才定义的卷积层中的这些层：

```
nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
nn.BatchNorm2d(128),
neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_function=surrogate.
↪ATan())
```

这 3 层网络，接收图片作为输入，输出脉冲，可以看作是编码器。

接下来，我们定义 3 层全连接网络，输出分类的结果。全连接层一般起到分类器的作用，使用 LIFNode 性能会更好。Fashion-MNIST 共有 10 类，因此输出层是 10 个神经元；为了减少过拟合，我们还使用了 layer.Dropout，关于它的更多信息可以参阅 API 文档。

```
self.fc = nn.Sequential(
    nn.Flatten(),
    layer.Dropout(0.5),
    nn.Linear(128 * 7 * 7, 128 * 3 * 3, bias=False),
    neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
```

(下页继续)

(续上页)

```
    layer.Dropout(0.5),
    nn.Linear(128 * 3 * 3, 128, bias=False),
    neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
    nn.Linear(128, 10, bias=False),
    neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
)
```

接下来，定义前向传播。前向传播非常简单，先经过卷积，在经过全连接即可：

```
def forward(self, x):
    return self.fc(self.conv(x))
```

1.6.2 避免重复计算

我们可以直接训练这个网络，就像之前的 MNIST 分类那样：

```
for img, label in train_data_loader:
    img = img.to(device)
    label = label.to(device)
    label_one_hot = F.one_hot(label, 10).float()

    optimizer.zero_grad()

    # 运行 T 个时长，out_spikes_counter 是 shape=[batch_size, 10] 的 tensor
    # 记录整个仿真时长内，输出层的 10 个神经元的脉冲发放次数
    for t in range(T):
        if t == 0:
            out_spikes_counter = net(encoder(img).float())
        else:
            out_spikes_counter += net(encoder(img).float())

    # out_spikes_counter / T 得到输出层 10 个神经元在仿真时长内的脉冲发放频率
    out_spikes_counter_frequency = out_spikes_counter / T

    # 损失函数为输出层神经元的脉冲发放频率，与真实类别的 MSE
    # 这样的损失函数会使，当类别 i 输入时，输出层中第 i 个神经元的脉冲发放频率趋近 1，而其他神经元的脉
冲发放频率趋近 0
    loss = F.mse_loss(out_spikes_counter_frequency, label_one_hot)
    loss.backward()
    optimizer.step()
```

(下页继续)

(续上页)

```
    # 优化一次参数后，需要重置网络的状态，因为 SNN 的神经元是有“记忆”的
    functional.reset_net(net)
```

但我们如果重新审视网络的结构，可以发现，有一些计算是重复的：对于网络的前 2 层，即下面代码中的高亮部分：

```
self.conv = nn.Sequential(
        nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(128),
        neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
        nn.MaxPool2d(2, 2),  # 14 * 14

        nn.Conv2d(128, 128, kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(128),
        neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
        nn.MaxPool2d(2, 2)  # 7 * 7
    )
```

这 2 层接收的输入图片，并不随 t 变化，但在 for 循环中，每次 img 都会重新经过这 2 层的计算，得到相同的输出。我们提取出这些层，同时将时间上的循环封装进网络本身，方便计算。新的网络结构完整定义为：

```
class Net(nn.Module):
    def __init__(self, tau, T, v_threshold=1.0, v_reset=0.0):
        super().__init__()
        self.T = T

        self.static_conv = nn.Sequential(
            nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(128),
        )

        self.conv = nn.Sequential(
            neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
            nn.MaxPool2d(2, 2),  # 14 * 14

            nn.Conv2d(128, 128, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(128),
            neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
            nn.MaxPool2d(2, 2)  # 7 * 7
```

(下页继续)

(续上页)

```
        )
        self.fc = nn.Sequential(
            nn.Flatten(),
            layer.Dropout(0.7),
            nn.Linear(128 * 7 * 7, 128 * 3 * 3, bias=False),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset,␣
↪surrogate_function=surrogate.ATan()),
            layer.Dropout(0.7),
            nn.Linear(128 * 3 * 3, 128, bias=False),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset,␣
↪surrogate_function=surrogate.ATan()),
            nn.Linear(128, 10, bias=False),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset,␣
↪surrogate_function=surrogate.ATan()),
        )


    def forward(self, x):
        x = self.static_conv(x)

        out_spikes_counter = self.fc(self.conv(x))
        for t in range(1, self.T):
            out_spikes_counter += self.fc(self.conv(x))

        return out_spikes_counter / self.T
```

对于输入是不随时间变化的 SNN，虽然 SNN 整体是有状态的，但网络的前几层可能没有状态，我们可以单独提取出这些层，将它们放到在时间上的循环之外，避免额外计算。

1.6.3 训练网络

完整的代码位于*spikingjelly.clock_driven.examples.conv_fashion_mnist*。也可以通过命令行直接运行。会将训练过程中测试集正确率最高的网络保存在 `tensorboard` 日志文件的同级目录下。实验机器使用 *Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz* 的 CPU 和 *GeForce RTX 2080 Ti* 的 GPU。

```
>>> from spikingjelly.clock_driven.examples import conv_fashion_mnist
>>> conv_fashion_mnist.main()
输入运行的设备，例如“cpu”或“cuda:0”
 input device, e.g., "cpu" or "cuda:0": cuda:9
输入保存 Fashion MNIST 数据集的位置，例如“./”
 input root directory for saving Fashion MNIST dataset, e.g., "./": ./fmnist
输入 batch_size，例如“64”
 input batch_size, e.g., "64": 128
```

(下页继续)

(续上页)

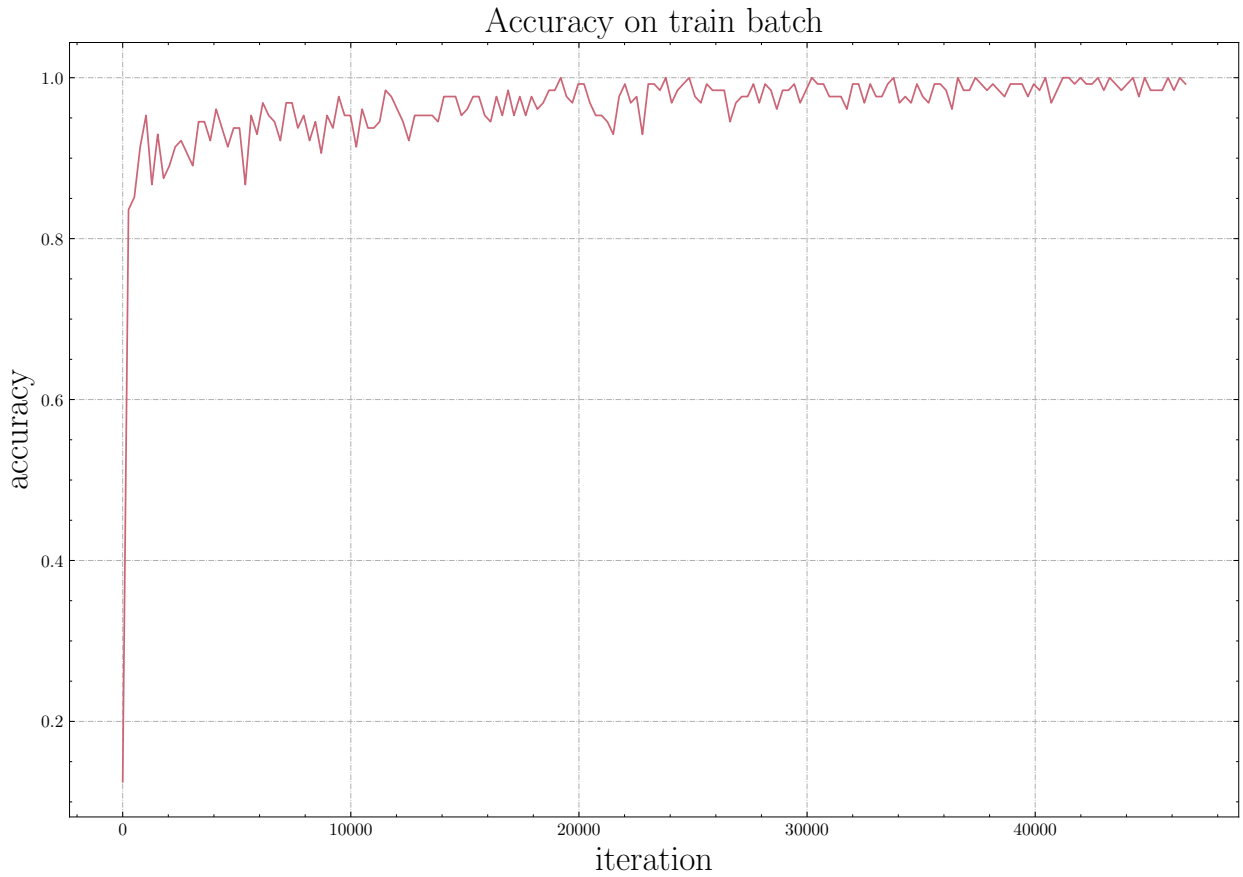
```
输入学习率，例如 “1e-3”
 input learning rate, e.g., "1e-3": 1e-3
输入仿真时长，例如 “8”
 input simulating steps, e.g., "8": 8
输入 LIF 神经元的时间常数 tau，例如 “2.0”
 input membrane time constant, tau, for LIF neurons, e.g., "2.0": 2.0
输入训练轮数，即遍历训练集的次数，例如 “100”
 input training epochs, e.g., "100": 100
输入保存 tensorboard 日志文件的位置，例如 “./”
 input root directory for saving tensorboard logs, e.g., "./": ./logs_conv_fashion_
↪mnist
saving net...
saved
epoch=0, t_train=41.182421264238656, t_test=2.5504338955506682, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_
↪fashion_mnist, max_test_accuracy=0.8704, train_times=468
saving net...
saved
epoch=1, t_train=40.93981215544045, t_test=2.538706629537046, device=cuda:0, dataset_
↪dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_fashion_
↪mnist, max_test_accuracy=0.8928, train_times=936
saving net...
saved
epoch=2, t_train=40.86129532009363, t_test=2.5383697943761945, device=cuda:0, dataset_
↪dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_fashion_
↪mnist, max_test_accuracy=0.899, train_times=1404
saving net...
saved

...

epoch=95, t_train=40.98498909268528, t_test=2.558146824128926, device=cuda:0, dataset_
↪dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_fashion_
↪mnist, max_test_accuracy=0.9425, train_times=44928
saving net...
saved
epoch=96, t_train=41.19765609316528, t_test=2.6626883540302515, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_
↪fashion_mnist, max_test_accuracy=0.9426, train_times=45396
saving net...
saved
epoch=97, t_train=41.10238983668387, t_test=2.553960849530995, device=cuda:0, dataset_
↪dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_fashion_
↪mnist, max_test_accuracy=0.9427, train_times=45864
```

(下页继续)

(续上页)

```
saving net...
saved
epoch=98, t_train=40.89284007716924, t_test=2.5465594390407205, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_
↪fashion_mnist, max_test_accuracy=0.944, train_times=46332
epoch=99, t_train=40.843392613343894, t_test=2.557370903901756, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_
↪fashion_mnist, max_test_accuracy=0.944, train_times=46800
```

运行 100 轮训练后，训练 batch 和测试集上的正确率如下：



在训练 100 个 epoch 后，最高测试集正确率可以达到 94.4%，对于 SNN 而言是非常不错的性能，仅仅略低于 Fashion-MNIST 的 BenchMark 中使用 Normalization, random horizontal flip, random vertical flip, random translation, random rotation 的 ResNet18 的 94.9% 正确率。

1.6.4 可视化编码器

正如我们在前文中所述，直接将数据送入 SNN，则首个脉冲神经元层及其之前的层，可以看作是一个可学习的编码器。具体而言，是我们的网络中如下所示的高亮部分：

```
class Net(nn.Module):
    def __init__(self, tau, T, v_threshold=1.0, v_reset=0.0):
        ...
        self.static_conv = nn.Sequential(
            nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(128),
        )

        self.conv = nn.Sequential(
            neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
        ...
```

现在让我们来查看一下，训练好的编码器，编码效果如何。让我们新建一个 python 文件，导入相关的模块，并重新定义一个 `batch_size=1` 的数据加载器，因为我们想要一张图片一张图片的查看：

```
from matplotlib import pyplot as plt
import numpy as np
from spikingjelly.clock_driven.examples.conv_fashion_mnist import Net
from spikingjelly import visualizing
import torch
import torch.nn as nn
import torchvision

test_data_loader = torch.utils.data.DataLoader(
    dataset=torchvision.datasets.FashionMNIST(
        root=dataset_dir,
        train=False,
        transform=torchvision.transforms.ToTensor(),
        download=True),
    batch_size=1,
    shuffle=True,
    drop_last=False)
```

从保存网络的位置，即 `log_dir` 目录下，加载训练好的网络，并提取出编码器。在 CPU 上运行即可：

```
net = torch.load('./logs_conv_fashion_mnist/net_max_acc.pt', 'cpu')
encoder = nn.Sequential(
    net.static_conv,
    net.conv[0]
)
encoder.eval()
```

接下来，从数据集中抽取一张图片，送入编码器，并查看输出脉冲的累加值 $\sum_t S_t$。为了显示清晰，我们还对输出的 `feature_map` 的像素值做了归一化，将数值范围线性变换到 `[0, 1]`。

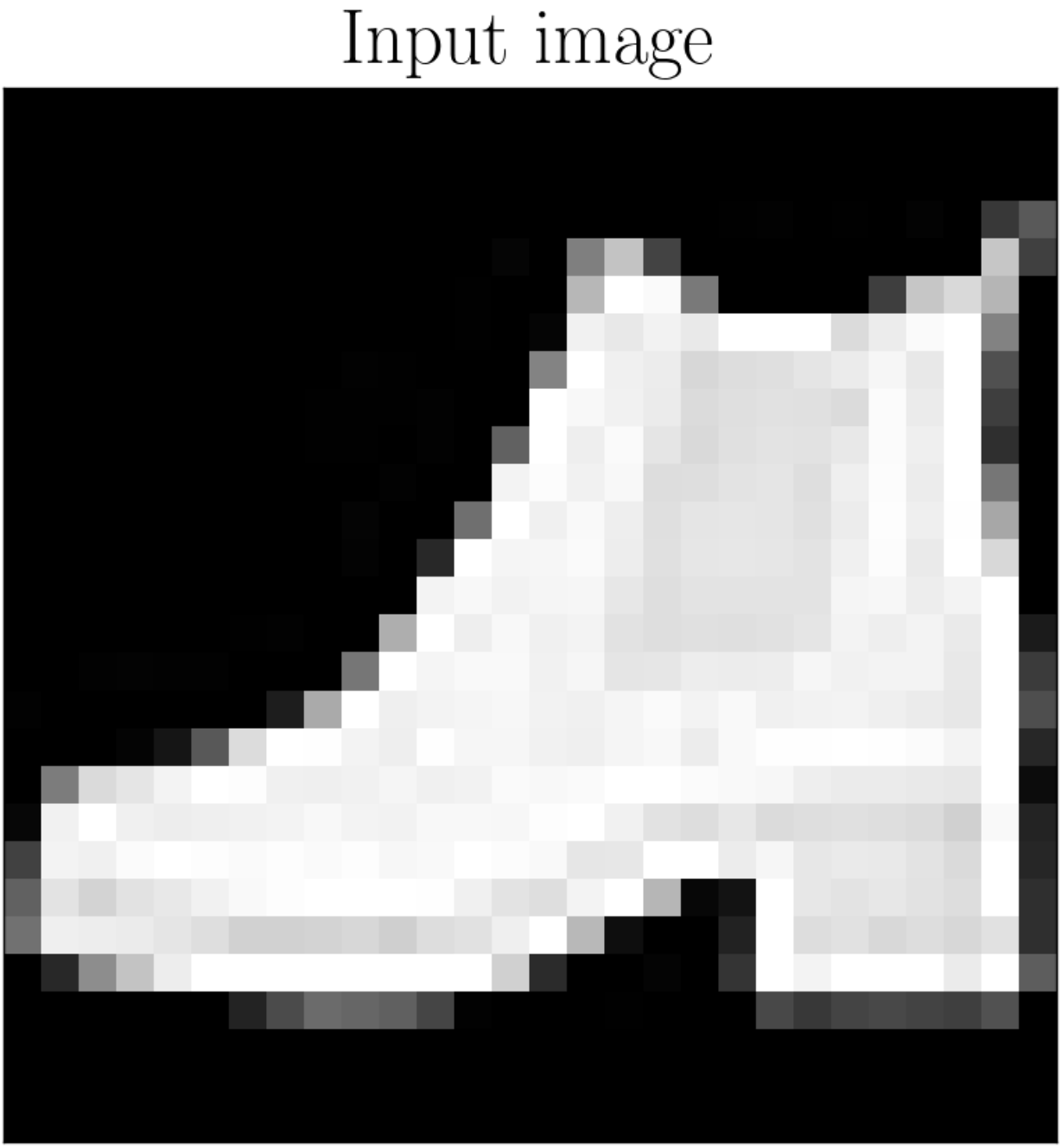
```
with torch.no_grad():
    # 每遍历一次全部数据集，就在测试集上测试一次
    for img, label in test_data_loader:
        fig = plt.figure(dpi=200)
        plt.imshow(img.squeeze().numpy(), cmap='gray')
        # 注意输入到网络的图片尺寸是 ``[1, 1, 28, 28]``，第 0 个维度是 ``batch``，第 1 个维度
是 ``channel``
        # 因此在调用 ``imshow`` 时，先使用 ``squeeze()`` 将尺寸变成 ``[28, 28]``
        plt.title('Input image', fontsize=20)
        plt.xticks([])
        plt.yticks([])
        plt.show()
```

(下页继续)

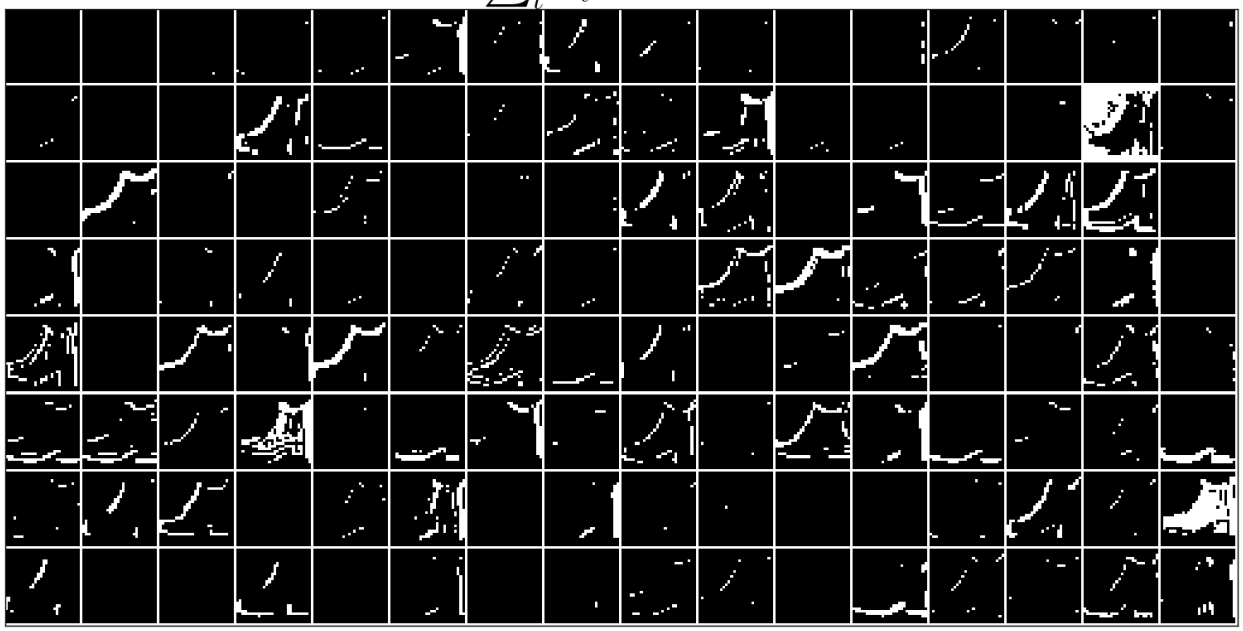
(续上页)

```
        out_spikes = 0
        for t in range(net.T):
            out_spikes += encoder(img).squeeze()
            # encoder(img) 的尺寸是 ``[1, 128, 28, 28]``, 同样使用 ``squeeze()`` 变换尺寸为 ``[128, 28, 28]``
            if t == 0 or t == net.T - 1:
                out_spikes_c = out_spikes.clone()
                for i in range(out_spikes_c.shape[0]):
                    if out_spikes_c[i].max().item() > out_spikes_c[i].min().item():
                        # 对每个 feature map 做归一化, 使显示更清晰
                        out_spikes_c[i] = (out_spikes_c[i] - out_spikes_c[i].min()) / (out_spikes_c[i].max() - out_spikes_c[i].min())
                visualizing.plot_2d_spiking_feature_map(out_spikes_c, 8, 16, 1, None)
                plt.title('$\\sum_{t} S_{t}$ at $t = ' + str(t) + '$', fontsize=20)
                plt.show()
```

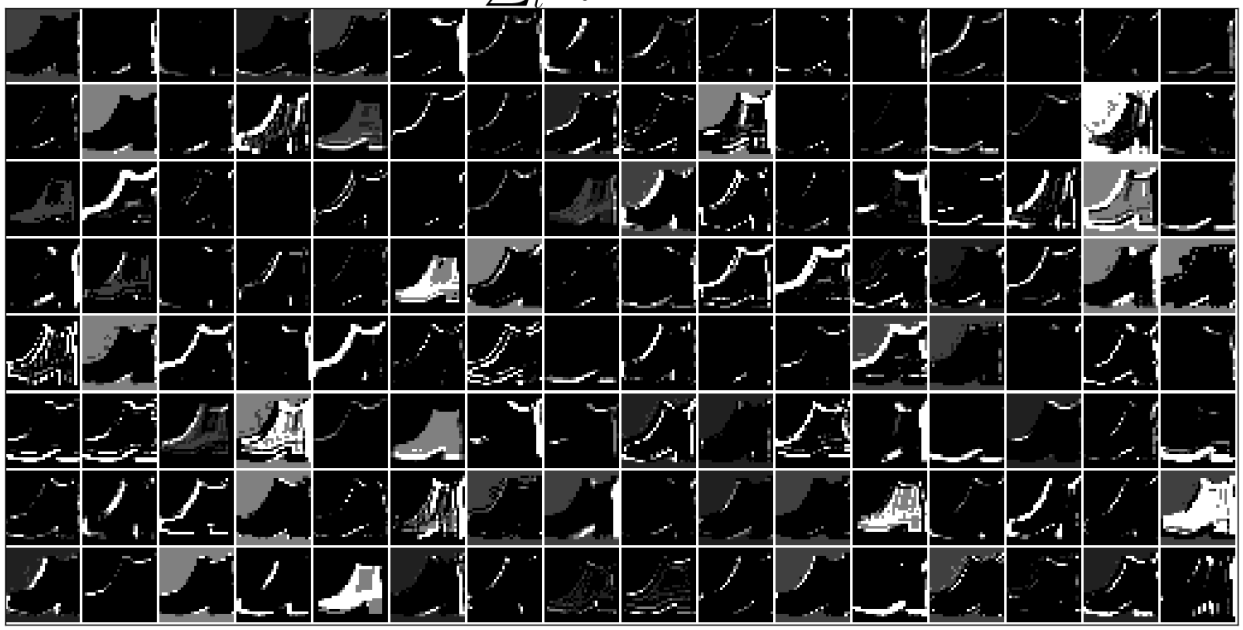
下面展示 2 个输入图片，以及在最开始 t=0 和最后 t=7 时刻的编码器输出的累计脉冲 $\sum_t S_t$：

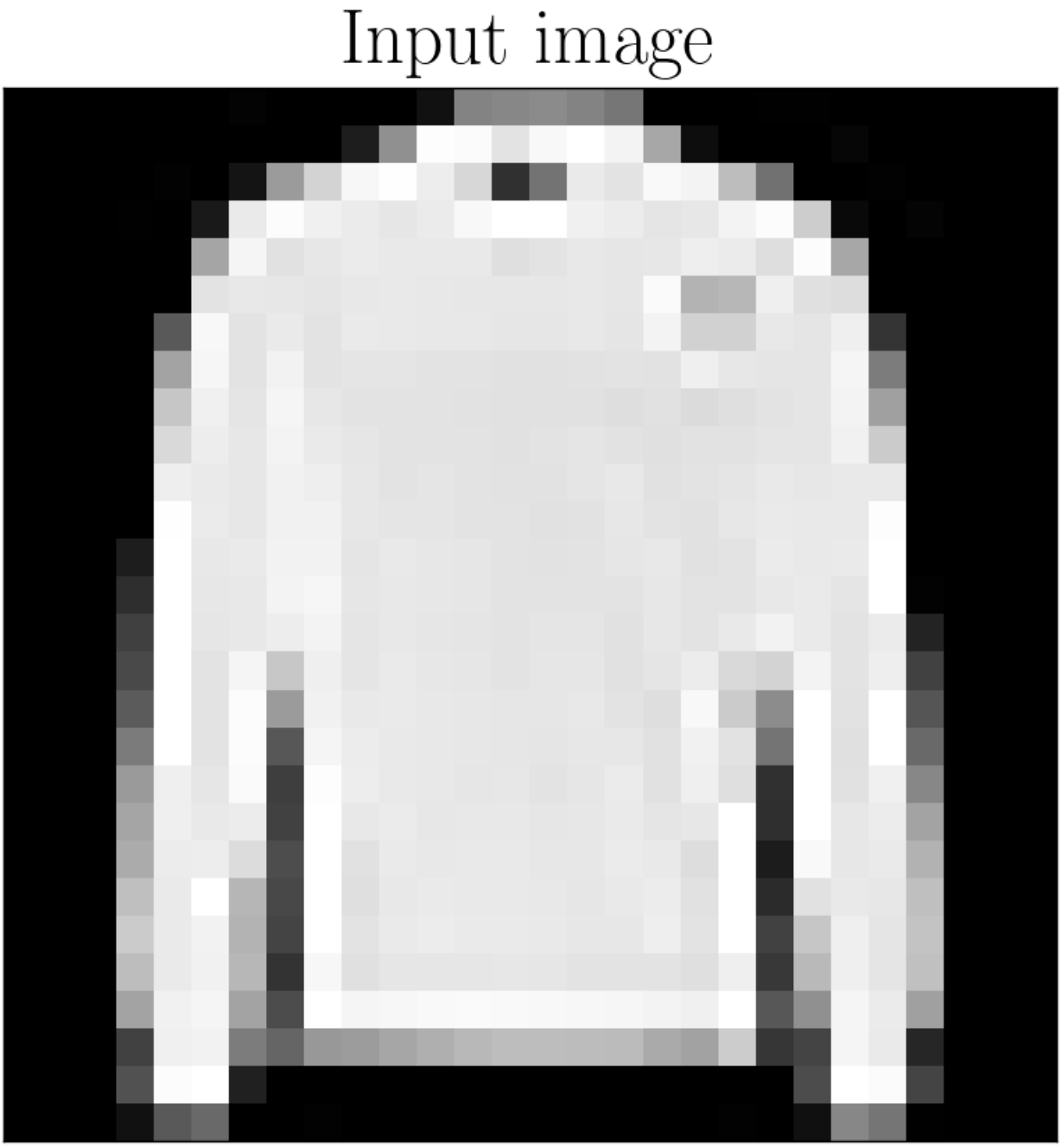

Input image

$\sum_t S_t$ at $t = 0$

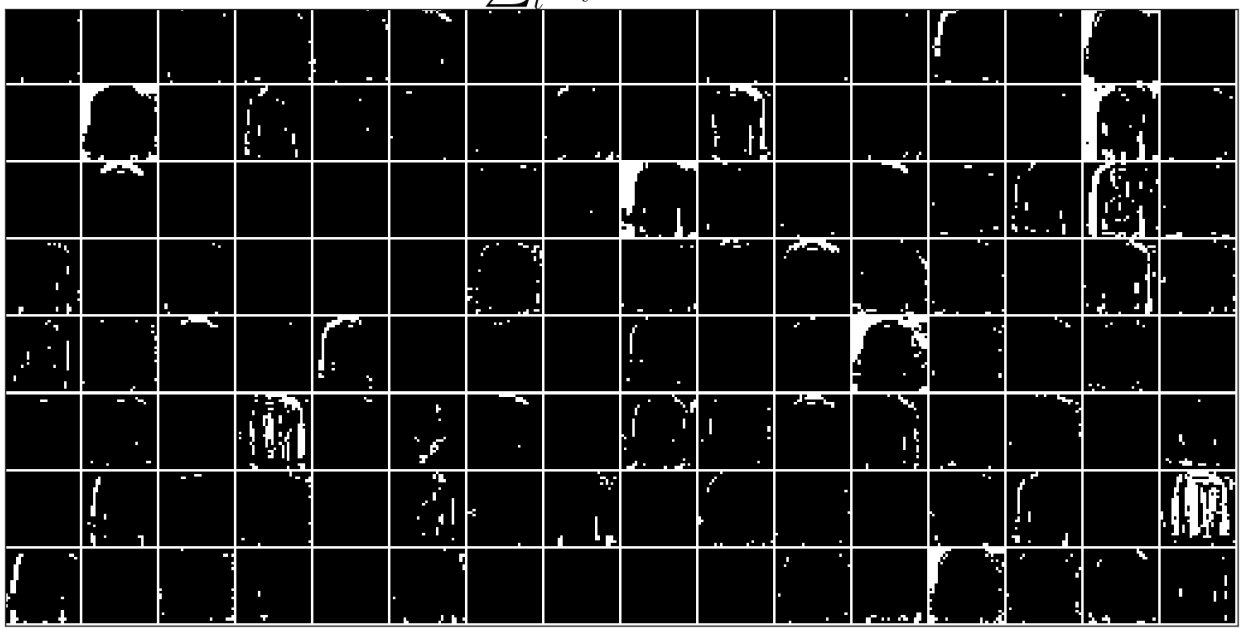


$\sum_t S_t$ at $t = 7$

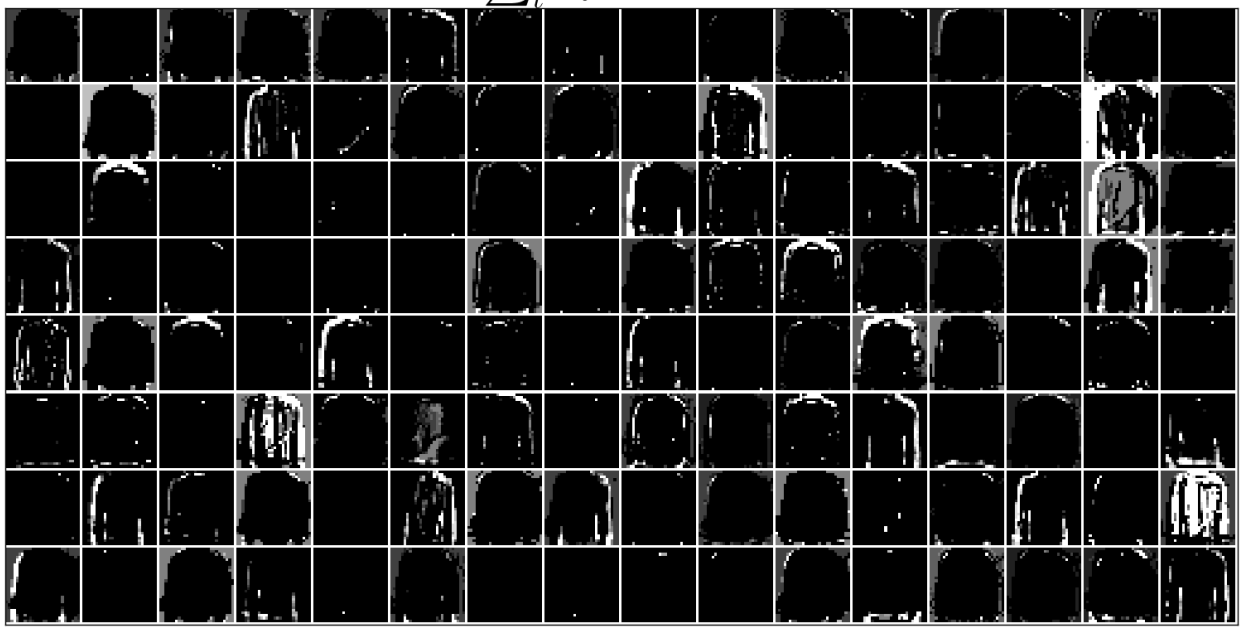



Input image

$\sum_t S_t$ at $t = 0$



$\sum_t S_t$ at $t = 7$



观察可以发现，编码器的累计输出脉冲 $\sum_t S_t$ 非常接近原图像的轮廓，表面这种自学习的脉冲编码器，有很强的编码能力。

1.7 ANN 转换 SNN

本教程作者：DingJianhao, fangwei123456

本节教程主要关注 `spikingjelly.clock_driven.ann2snn`，介绍如何将训练好的 ANN 转换 SNN，并且在 SpikingJelly 框架上进行仿真。

目前实现了两套实现：基于 ONNX 和基于 PyTorch，在框架中被称为 ONNX kernel 和 PyTorch kernel。但是这两套实现各有特点，ONNX kernel 的实现更加通用，支持更加复杂的拓扑结构（例如 ResNet）；PyTorch kernel 主要是为了简单测试，支持的模块比较有限且在现有配置下可能有很多 bug。更多模块可以通过 ONNX 拓展，用户可自行实现…

1.7.1 ANN 转换 SNN 的理论基础

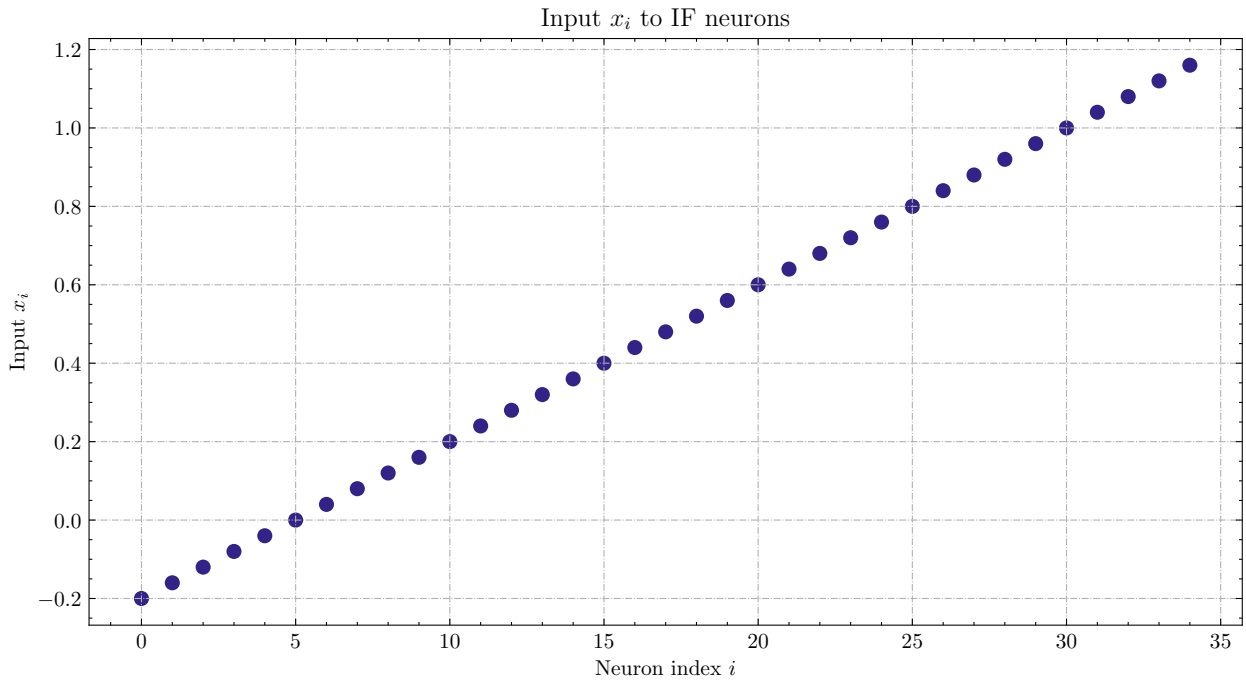
SNN 相比于 ANN，产生的脉冲是离散的，这有利于高效的通信。在 ANN 大行其道的今天，SNN 的直接训练需要较多资源。自然我们会想到使用现在非常成熟的 ANN 转换到 SNN，希望 SNN 也能有类似的表现。这就牵扯到如何搭建起 ANN 和 SNN 桥梁的问题。现在 SNN 主流的方式是采用频率编码，因此对于输出层，我们会用神经元输出脉冲数来判断类别。发放率和 ANN 有没有关系呢?

幸运的是，ANN 中的 ReLU 神经元非线性激活和 SNN 中 IF 神经元 (采用减去阈值 $V_{threshold}$ 方式重置) 的发放率有着极强的相关性，我们可以借助这个特性来进行转换。这里说的神经元更新方式，也就是 时间驱动教程 中提到的 Soft 方式。

实验：IF 神经元脉冲发放频率和输入的关系

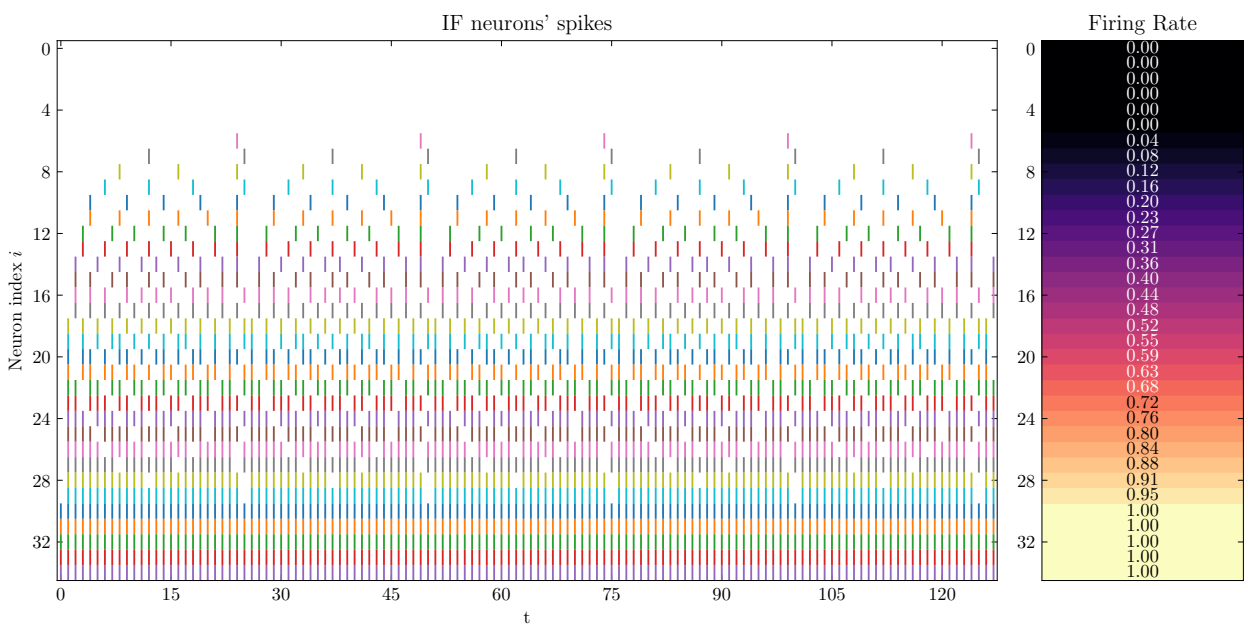
我们给与恒定输入到 IF 神经元，观察其输出脉冲和脉冲发放频率。首先导入相关的模块，新建 IF 神经元层，确定输入并画出每个 IF 神经元的输入 x_i：

```
import torch
from spikingjelly.clock_driven import neuron
from spikingjelly import visualizing
from matplotlib import pyplot as plt
import numpy as np

plt.rcParams['figure.dpi'] = 200
if_node = neuron.IFNode(v_reset=None, monitor_state=True)
T = 128
x = torch.arange(-0.2, 1.2, 0.04)
plt.scatter(torch.arange(x.shape[0]), x)
plt.title('Input $x_{i}$ to IF neurons')
plt.xlabel('Neuron index $i$')
plt.ylabel('Input $x_{i}$')
plt.grid(linestyle='-.')
plt.show()
```



接下来，将输入送入到 IF 神经元层，并运行 `T=128` 步，观察各个神经元发放的脉冲、脉冲发放频率：

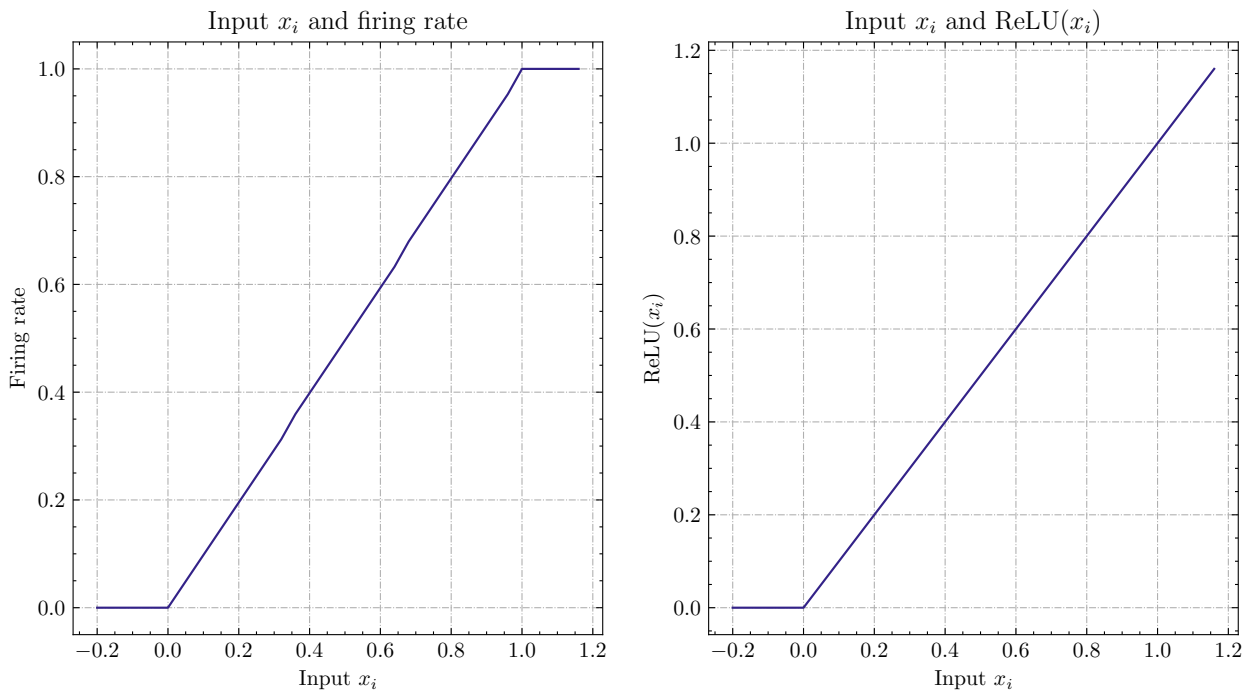
```
for t in range(T):
    if_node(x)
out_spikes = np.asarray(if_node.monitor['s']).T
visualizing.plot_1d_spikes(out_spikes, 'IF neurons\' spikes and firing rates', 't',
↪'Neuron index $i$')
plt.show()
```



可以发现，脉冲发放的频率在一定范围内，与输入 x_i 的大小成正比。

接下来，让我们画出 IF 神经元脉冲发放频率和输入 x_i 的曲线，并与 ReLU(x_i) 对比：

```
plt.subplot(1, 2, 1)
firing_rate = np.mean(out_spikes, axis=1)
plt.plot(x, firing_rate)
plt.title('Input $x_{i}$ and firing rate')
plt.xlabel('Input $x_{i}$')
plt.ylabel('Firing rate')
plt.grid(linestyle='-.')

plt.subplot(1, 2, 2)
plt.plot(x, x.relu())
plt.title('Input $x_{i}$ and ReLU($x_{i}$)')
plt.xlabel('Input $x_{i}$')
plt.ylabel('ReLU($x_{i}$)')
plt.grid(linestyle='-.')
plt.show()
```



可以发现，两者的曲线几乎一致。需要注意的是，脉冲频率不可能高于 1，因此 IF 神经元无法拟合 ANN 中 ReLU 的输入大于 1 的情况。

理论证明

文献[1] 对 ANN 转 SNN 提供了解析的理论基础。理论说明，SNN 中的 IF 神经元是 ReLU 激活函数在时间上的无偏估计器。

针对神经网络第一层即输入层，讨论 SNN 神经元的发放率 r 和对应 ANN 中激活的关系。假定输入恒定为 $z \in [0, 1]$。对于采用减法重置的 IF 神经元，其膜电位 V 随时间变化为：

$$V_t = V_{t-1} + z - V_{threshold}\theta_t$$

其中： $V_{threshold}$ 为发放阈值，通常设为 1.0。θ_t 为输出脉冲。T 时间步内的平均发放率可以通过对膜电位求和得到：

$$\sum_{t=1}^{T} V_t = \sum_{t=1}^{T} V_{t-1} + zT - V_{threshold}\sum_{t=1}^{T}\theta_t$$

将含有 V_t 的项全部移项到左边，两边同时除以 T：

$$\frac{V_T - V_0}{T} = z - V_{threshold}\frac{\sum_{t=1}^{T}\theta_t}{T} = z - V_{threshold}\frac{N}{T}$$

其中 N 为 T 时间步内脉冲数，$\frac{N}{T}$ 就是发放率 r。利用 $z = V_{threshold}a$ 即：

$$r = a - \frac{V_T - V_0}{TV_{threshold}}$$

故在仿真时间步 T 无限长情况下：

$$r = a(a > 0)$$

类似地，针对神经网络更高层，文献[?] 进一步说明层间发放率满足：

$$r^l = W^l r^{l-1} + b^l - \frac{V_T^l}{TV_{threshold}}$$

详细的说明见文献[?]。ann2snn 中的方法也主要来自文献[?]

1.7.2 转换和仿真

具体地，进行前馈 ANN 转 SNN 主要有两个步骤：即模型分析（英文：parse，直译：句法分析）和仿真模拟。

模型分析

模型分析主要解决两个问题：

1. ANN 为了快速训练和收敛提出了批归一化（Batch Normalization）。批归一化旨在将 ANN 输出归一化到 0 均值，这与 SNN 的特性相违背。因此，需要将 BN 的参数吸收到前面的参数层中（Linear、Conv2d）

[1] Rueckauer B, Lungu I-A, Hu Y, Pfeiffer M and Liu S-C (2017) Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification. Front. Neurosci. 11:682.

2. 根据转换理论，ANN 的每层输入输出需要被限制在 [0,1] 范围内，这就需要对参数进行缩放（模型归一化）

◆ BatchNorm 参数吸收

假定 BatchNorm 的参数为 γ (`BatchNorm.weight`)，β (`BatchNorm.bias`)，μ (`BatchNorm.running_mean`) ，σ (`BatchNorm.running_var`，$\sigma = \sqrt{\text{running\_var}}$)。具体参数定义详见 torch.nn.BatchNorm1d 。参数模块（例如 Linear）具有参数 W 和 b 。BatchNorm 参数吸收就是将 BatchNorm 的参数通过运算转移到参数模块的 W 中，使得数据输入新模块的输出和有 BatchNorm 时相同。对此，新模型的 $\bar{W}$ 和 $\bar{b}$ 公式表示为：

$$\bar{W} = \frac{\gamma}{\sigma} W$$

$$\bar{b} = \frac{\gamma}{\sigma}(b - \mu) + \beta$$

◆ 模型归一化

对于某个参数模块，假定得到了其输入张量和输出张量，其输入张量的最大值为 λ_{pre} ，输出张量的最大值为 λ 那么，归一化后的权重 $\hat{W}$ 为：

$$\hat{W} = W * \frac{\lambda_{pre}}{\lambda}$$

归一化后的偏置 $\hat{b}$ 为：

$$\hat{b} = \frac{b}{\lambda}$$

ANN 每层输出的分布虽然服从某个特定分布，但是数据中常常会存在较大的离群值，这会导致整体神经元发放率降低。为了解决这一问题，鲁棒归一化将缩放因子从张量的最大值调整为张量的 p 分位点。文献中推荐的分位点值为 99.9。

到现在为止，我们对神经网络做的操作，在数值上是完全等价的。当前的模型表现应该与原模型相同。

模型仿真

仿真前，我们需要将原模型中的 ReLU 激活函数变为 IF 神经元。对于 ANN 中的平均池化，我们需要将其转化为空间下采样。由于 IF 神经元可以等效 ReLU 激活函数。空间下采样后增加 IF 神经元与否对结果的影响极小。对于 ANN 中的最大池化，目前没有非常理想的方案。目前的最佳方案为使用基于动量累计脉冲的门控函数控制脉冲通道[7]。当然在 ONNX kernel 中没有用，不过我们在 “ann2snn.modules“依然有实现。还有文献提出使用空间下采样替代 Maxpool2d。此处我们依然推荐使用 avgpool2d。

仿真时，依照转换理论，SNN 需要输入恒定的模拟输入。使用 Poisson 编码器将会带来准确率的降低。Poisson 编码和恒定输入方式均已实现，感兴趣可通过配置进行不同实验。

实现与可选配置

ann2snn 框架在 2020 年 12 月进行一次较大更新。最大改动就是将参数配置回归到了模块参数，并且尽可能考虑到了用户对灵活度和渐变操作的需求。这里我们将简单介绍一下这些类和方法。针对理论中提到的分析和仿真两大中心，设计了 parser 和 simulator 两大类。类的定义在 “spikingjelly.ann2snn.__init__“中。

◆ parser 类 1. 类初始化函数 - kernel：转换的 kernel。可选范围为’onnx’、’pytorch’，这将决定您使用的是 ONNX kernel 还是 PyTorch kernel - name：模型的名字，通常您可以取一个和任务、模型相关的名字，之后的文件夹生成将可能用到这个字符串 - z_norm：许多深度学习模型会存在数据标准化（Z normalization）。如果您 ANN 模型有这个操作，这个参数的数据格式为：(mean, std)，例如对于 CIFAR10，z_norm 可以为 ((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)) - log_dir：保存临时文件的文件夹，如没有此参数则会根据参数 name 和当前时间自动生成 - json：历史配置文件名。当您运行过一次 parser 后，程序会自动保存 json 文件到 log_dir，您可以使用 json 文件进行 parser 快速初始化

2. parse 函数 - channelwise: 如果为 “True“，则控制激活幅值的统计是 channelwise 的；否则，控制激活幅值的统计是 layerwise 的 - robust: 如果为 “True“，则控制激活幅值的统计是激活的 99.9 百分位；否则，控制激活幅值的统计是激活的最值 - user_methods：默认使用 “spikingjelly.ann2snn.kernel.onnx._o2p_converter“；当发现 ONNX kernel 遇到 ONNX 转换 PyTorch 的方法缺乏的时候，可以通过用户自定义函数的形式进行转换。函数接口可见 “spikingjelly.ann2snn.kernel.onnx._o2p_converter“的 staticmethods

◆ simulator 类 1. 类初始化参数 - snn：待仿真的转换后的 SNN - device：仿真的设备，支持单设备（输入为字符串）和多设备（输入为 list,set,tuple 类型）- name：模型的名字，通常您可以取一个和任务、模型相关的名字，之后的文件夹生成将可能用到这个字符串 - log_dir：保存临时文件的文件夹，如没有此参数则会根据参数 name 和当前时间自动生成 - encoder：编码器，可选范围为’constant’、’poisson’

2. simulate 函数 - data_loader：仿真的数据集的 dataloader - T：仿真时间 - canvas：plt.fig 类型，用于对仿真模型标量性能（例如准确率）的绘图 - online_drawer：如果为 “True“，则在线绘图；否则，仿真结束后绘图 - func_dict：用户可以通过自己定义标量性能函数实现绘图

除此之外，用户可以通过继承 simulate 类进行仿真器的功能细化。比如 “spikingjelly.ann2snn.__init__“实现了仿真分类任务的 “classify_simulator“

3. classify_simulator.simulate 函数除去继承的参数外，- ann_acc：ANN 转换前的分类准确率（0-1 间的小数）- fig_name: 仿真图像的名字 - step_max：如果为 “True“，则图像中标明推理过程中的最大准确率

1.7.3 识别 MNIST

现在我们使用 `ann2snn`，搭建一个简单卷积网络，对 MNIST 数据集进行分类。

首先定义我们的网络结构：

```
class ANN(nn.Module):
    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(
            nn.Conv2d(1, 32, 3, 1),
```

(下页继续)

(续上页)

```
            nn.BatchNorm2d(32, eps=1e-3),
            nn.ReLU(),
            nn.AvgPool2d(2, 2),

            nn.Conv2d(32, 32, 3, 1),
            nn.BatchNorm2d(32, eps=1e-3),
            nn.ReLU(),
            nn.AvgPool2d(2, 2),

            nn.Conv2d(32, 32, 3, 1),
            nn.BatchNorm2d(32, eps=1e-3),
            nn.ReLU(),
            nn.AvgPool2d(2, 2),

            nn.Flatten(),
            nn.Linear(32, 10),
            nn.ReLU()
        )

    def forward(self,x):
        x = self.network(x)
        return x
```

注意：如果遇到需要将 tensor 展开的情况，就在网络中定义一个 `nn.Flatten` 模块，在 forward 函数中需要使用定义的 Flatten 而不是 view 函数。

定义我们的超参数：

```
device = input('输入运行的设备，例如“cpu”或“cuda:0” \n input device, e.g., "cpu" or
↪"cuda:0": ')
dataset_dir = input('输入保存 MNIST 数据集的位置，例如“./” \n input root directory for␣
↪saving MNIST dataset, e.g., "./": ')
batch_size = int(input('输入 batch_size，例如“64” \n input batch_size, e.g., "64": '))
learning_rate = float(input('输入学习率，例如“1e-3” \n input learning rate, e.g., "1e-3
↪": '))
T = int(input('输入仿真时长，例如“100” \n input simulating steps, e.g., "100": '))
train_epoch = int(input('输入训练轮数，即遍历训练集的次数，例如“10” \n input training epochs,
↪ e.g., "10": '))
model_name = input('输入模型名字，例如“mnist” \n input model name, for log_dir generating␣
↪, e.g., "mnist": ')
```

之后的所有临时文件都会储存到文件夹中。

初始化数据加载器、网络、优化器、损失函数：

```
# 初始化网络
ann = ANN().to(device)
# 定义损失函数
loss_function = nn.CrossEntropyLoss()
# 使用 Adam 优化器
optimizer = torch.optim.Adam(ann.parameters(), lr=learning_rate, weight_decay=5e-4)
```

训练 ANN，并定期测试。训练时也可以使用 utils 中预先写好的训练程序：

```
for epoch in range(train_epoch):
    # 使用 utils 中预先写好的训练程序训练网络
    # 训练程序的写法和经典 ANN 中的训练也是一样的
    # Train the network using a pre-prepared code in ''utils''
    utils.train_ann(net=ann,
                    device=device,
                    data_loader=train_data_loader,
                    optimizer=optimizer,
                    loss_function=loss_function,
                    epoch=epoch
                    )
    # 使用 utils 中预先写好的验证程序验证网络输出
    # Validate the network using a pre-prepared code in ''utils''
    acc = utils.val_ann(net=ann,
                        device=device,
                        data_loader=test_data_loader,
                        epoch=epoch
                        )
    if best_acc <= acc:
        utils.save_model(ann, log_dir, model_name+'.pkl')
```

完整的代码位于 `ann2snn.examples.cnn_mnist.py`，在代码中我们还使用了 Tensorboard 来保存训练日志。可以直接在 Python 命令行运行它：

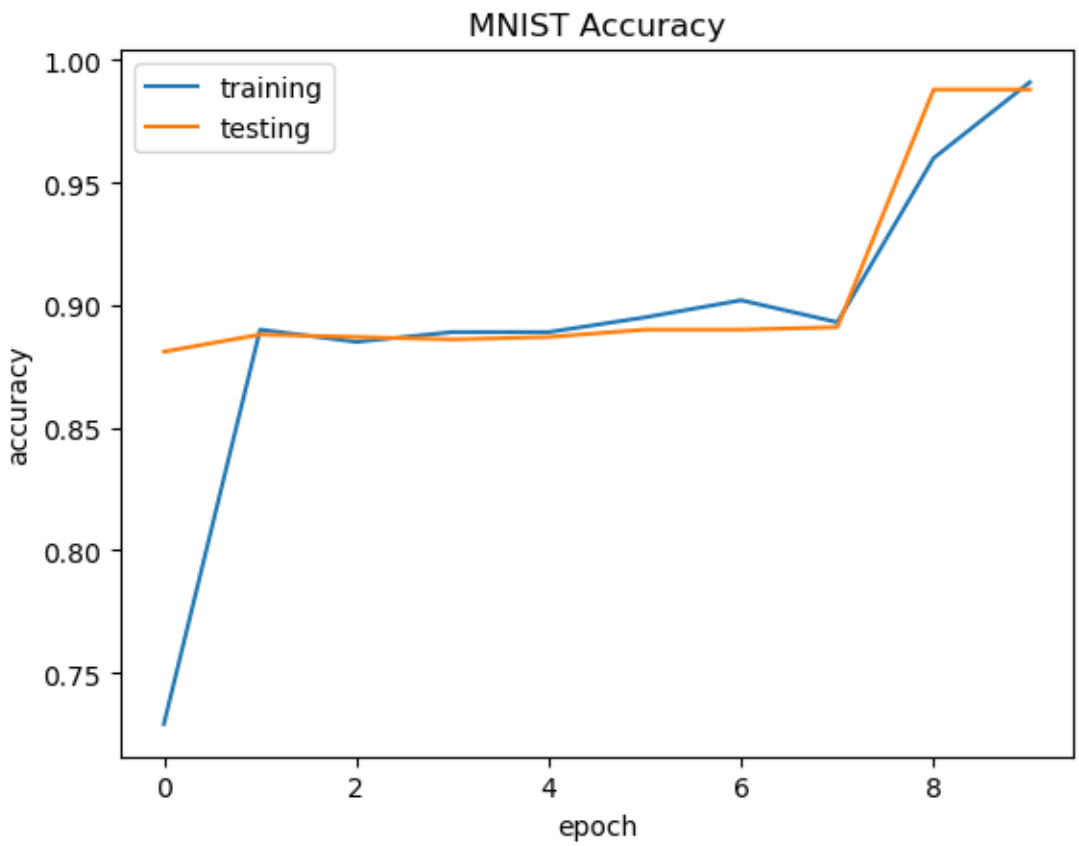
```
>>> import spikingjelly.clock_driven.ann2snn.examples.cnn_mnist as cnn_mnist
>>> cnn_mnist.main()
输入运行的设备，例如“cpu”或“cuda:0”
 input device, e.g., "cpu" or "cuda:0": cuda:15
输入保存 MNIST 数据集的位置，例如“./”
 input root directory for saving MNIST dataset, e.g., "./": ./mnist
输入 batch_size，例如“64”
 input batch_size, e.g., "64": 128
输入学习率，例如“1e-3”
 input learning rate, e.g., "1e-3": 1e-3
输入仿真时长，例如“100”
```

(下页继续)

(续上页)

```
 input simulating steps, e.g., "100": 100
输入训练轮数，即遍历训练集的次数，例如“10”
 input training epochs, e.g., "10": 10
输入模型名字，用于自动生成日志文档，例如“cnn_mnist”
 input model name, for log_dir generating , e.g., "cnn_mnist"

Epoch 0 [1/937] ANN Training Loss:2.252 Accuracy:0.078
Epoch 0 [101/937] ANN Training Loss:1.423 Accuracy:0.669
Epoch 0 [201/937] ANN Training Loss:1.117 Accuracy:0.773
Epoch 0 [301/937] ANN Training Loss:0.953 Accuracy:0.795
Epoch 0 [401/937] ANN Training Loss:0.865 Accuracy:0.788
Epoch 0 [501/937] ANN Training Loss:0.807 Accuracy:0.792
Epoch 0 [601/937] ANN Training Loss:0.764 Accuracy:0.795
Epoch 0 [701/937] ANN Training Loss:0.726 Accuracy:0.835
Epoch 0 [801/937] ANN Training Loss:0.681 Accuracy:0.880
Epoch 0 [901/937] ANN Training Loss:0.641 Accuracy:0.889
100%|██████████| 100/100 [00:00<00:00, 116.12it/s]
Epoch 0 [100/100] ANN Validating Loss:0.327 Accuracy:0.881
Save model to: cnn_mnist-XXXXX\cnn_mnist.pkl
......
```

示例中，这个模型训练 10 个 epoch。训练时测试集准确率变化情况如下：



最终达到 98.8% 的测试集准确率。

从训练集中，取出一部分数据，用于模型的归一化步骤。这里我们取 192 张图片。

```
# 加载用于归一化模型的数据
# Load the data to normalize the model
percentage = 0.004 # load 0.004 of the data
norm_data_list = []
for idx, (imgs, targets) in enumerate(train_data_loader):
    norm_data_list.append(imgs)
    if idx == int(len(train_data_loader) * percentage) - 1:
        break
norm_data = torch.cat(norm_data_list)
print('use %d imgs to parse' % (norm_data.size(0)))
```

调用 `ann2snn` 中的类 parser，并使用 ONNX kernel。

我们可以保存好我们转换好的 snn 模型，并且定义一个 plt.figure 用于绘图

```
torch.save(snn, os.path.join(log_dir,'snn-'+model_name+'.pkl'))
fig = plt.figure('simulator')
```

现在，我们定义用于 SNN 的仿真器。由于我们的任务是分类，选择类 “classify_simulator“

```
sim = classify_simulator(snn,
                         log_dir=log_dir + '/simulator',
                         device=simulator_device,
                         canvas=fig
                         )
sim.simulate(test_data_loader,
            T=T,
            online_drawer=True,
            ann_acc=ann_acc,
            fig_name=model_name,
            step_max=True
            )
```

模型仿真由于时间较长，我们设计了 tqdm 的进度条用于预估仿真时间。仿真结束时会有仿真器的 summary

```
simulator is working on the normal mode, device: cuda:0
100%|██████████| 100/100 [00:46<00:00,  2.15it/s]
--------------------simulator summary--------------------
time elapsed: 46.55072790000008 (sec)
-----------------------------------------------------------
```

通过最后的输出，可以知道，仿真器使用了 46.6s。转换后的 SNN 准确率可以从 simulator 文件夹中 plot.pdf 看到，最高的转换准确率为 98.51%。转换带来了 0.37% 的性能下降。通过增加推理时间可以减少转换损失。

1.8 强化学习 DQN

本教程作者：fangwei123456，lucifer2859

本节教程使用 SNN 重新实现 PyTorch 官方的 REINFORCEMENT LEARNING (DQN) TUTORIAL。请确保你已经阅读了原版教程和代码，因为本教程是对原教程的扩展。

1.8.1 更改输入

在 ANN 的实现中，直接使用 CartPole 的相邻两帧之差作为输入，然后使用 CNN 来提取特征。使用 SNN 实现，也可以用相同的方法，但目前的 Gym 若想得到帧数据，必须启动图形界面，不便于在无图形界面的服务器上进行训练。为了降低难度，我们将输入更改为 CartPole 的 Observation，即 Cart Position，Cart Velocity，Pole Angle 和 Pole Velocity At Tip，这是一个包含 4 个 float 元素的数组。训练的代码也需要做相应改动，将在下文展示。

输入已经更改为 4 个 float 元素的数组，记下来我们来定义 SNN。需要注意，在 Deep Q Learning 中，神经网络充当 Q 函数，而 Q 函数的输出应该是一个连续值。这意味着我们的 SNN 最后一层不能输出脉冲，否则我们的 Q 函数永远都输出 0 和 1，使用这样的 Q 函数，效果会非常差。让 SNN 输出连续值的方法有很多，之

前教程中的分类任务，网络最终的输出是输出层的脉冲发放频率，它是累计所有时刻的输出脉冲，再除以仿真时长得到的。在这个任务中，如果我们也使用脉冲发放频率，效果可能会很差，因此脉冲发放频率并不是非常“连续”：仿真 T 步，可能的脉冲发放频率取值只能是 $0, \frac{1}{T}, \frac{2}{T}, ..., 1$。

我们使用另一种常用的使 SNN 输出浮点值的方法：将神经元的阈值设置成无穷大，使其不发放脉冲，用神经元最后时刻的电压作为输出值。神经元实现这种神经元非常简单，只需要继承已有神经元，重写 `forward` 函数即可。LIF 神经元的电压不像 IF 神经元那样是简单的积分，因此我们使用 LIF 神经元来改写：

```
class NonSpikingLIFNode(neuron.LIFNode):
        class NonSpikingLIFNode(neuron.LIFNode):
    def forward(self, dv: torch.Tensor):
        self.neuronal_charge(dv)
        # self.neuronal_fire()
        # self.neuronal_reset()
        return self.v
```

接下来，搭建我们的 Deep Q Spiking Network，网络的结构非常简单，全连接-IF 神经元-全连接-NonSpikingLIF 神经元，全连接-IF 神经元起到编码器的作用，而全连接-NonSpikingLIF 神经元则可以看作一个决策器：

```
class DQSN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, T=16):
        super().__init__()
        self.fc = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            neuron.IFNode(),
            nn.Linear(hidden_size, output_size),
            NonSpikingLIFNode(tau=2.0)
        )

        self.T = T

    def forward(self, x):
        for t in range(self.T):
            self.fc(x)

        return self.fc[-1].v
```

1.8.2 训练网络

训练部分的代码，与 ANN 版本几乎相同。需要注意的是，ANN 使用两帧之差作为输入，而我们使用 env 返回的 Observation 作为输入。

ANN 的原始代码为：

```
for i_episode in range(num_episodes):
    # Initialize the environment and state
    env.reset()
    last_screen = get_screen()
    current_screen = get_screen()
    state = current_screen - last_screen
    for t in count():
        # Select and perform an action
        action = select_action(state)
        _, reward, done, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device)

        # Observe new state
        last_screen = current_screen
        current_screen = get_screen()
        if not done:
            next_state = current_screen - last_screen
        else:
            next_state = None

        # Store the transition in memory
        memory.push(state, action, next_state, reward)

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the target network)
        optimize_model()
        if done:
            episode_durations.append(t + 1)
            plot_durations()
            break
    # Update the target network, copying all weights and biases in DQN
    if i_episode % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict())
```

SNN 的训练代码如下，我们会保存训练过程中使得奖励最大的模型参数：

```
for i_episode in range(num_episodes):
    # Initialize the environment and state
    env.reset()
    state = torch.zeros([1, n_states], dtype=torch.float, device=device)

    total_reward = 0

    for t in count():
        action = select_action(state, steps_done)
        steps_done += 1
        next_state, reward, done, _ = env.step(action.item())
        total_reward += reward
        next_state = torch.from_numpy(next_state).float().to(device).unsqueeze(0)
        reward = torch.tensor([reward], device=device)

        if done:
            next_state = None

        memory.push(state, action, next_state, reward)

        state = next_state
        if done and total_reward > max_reward:
            max_reward = total_reward
            torch.save(policy_net.state_dict(), max_pt_path)
            print(f'max_reward={max_reward}, save models')

        optimize_model()

        if done:
            print(f'Episode: {i_episode}, Reward: {total_reward}')
            writer.add_scalar('Spiking-DQN-state-' + env_name + '/Reward', total_
↪reward, i_episode)
            break

    if i_episode % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict())
```

另外一个需要注意的地方是，SNN 是有状态的，因此每次前向传播后，不要忘了将网络 `reset`。涉及到的代码如下：

```
def select_action(state, steps_done):
    ...
    if sample > eps_threshold:
        with torch.no_grad():
```

(下页继续)

(续上页)

```
            ac = policy_net(state).max(1)[1].view(1, 1)
            functional.reset_net(policy_net)
    ...

def optimize_model():
    ...
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0].
↪detach()
    functional.reset_net(target_net)
    ...
    optimizer.step()
    functional.reset_net(policy_net)
```

完整的代码可见于 clock_driven/examples/Spiking_DQN_state.py。可以从命令行直接启动训练：

```
>>> from spikingjelly.clock_driven.examples import Spiking_DQN_state
>>> Spiking_DQN_state.train(use_cuda=False, model_dir='./model/CartPole-v0', log_dir=
↪'./log', env_name='CartPole-v0', hidden_size=256, num_episodes=500, seed=1)
...
Episode: 509, Reward: 715
Episode: 510, Reward: 3051
Episode: 511, Reward: 571
complete
state_dict path is./ policy_net_256.pt
```

1.8.3 用训练好的网络玩 CartPole

我们从服务器上下载训练过程中使奖励最大的模型 `policy_net_256_max.pt`，在有图形界面的本机上运行 `play` 函数，用训练了 512 次的网络来玩 CartPole：

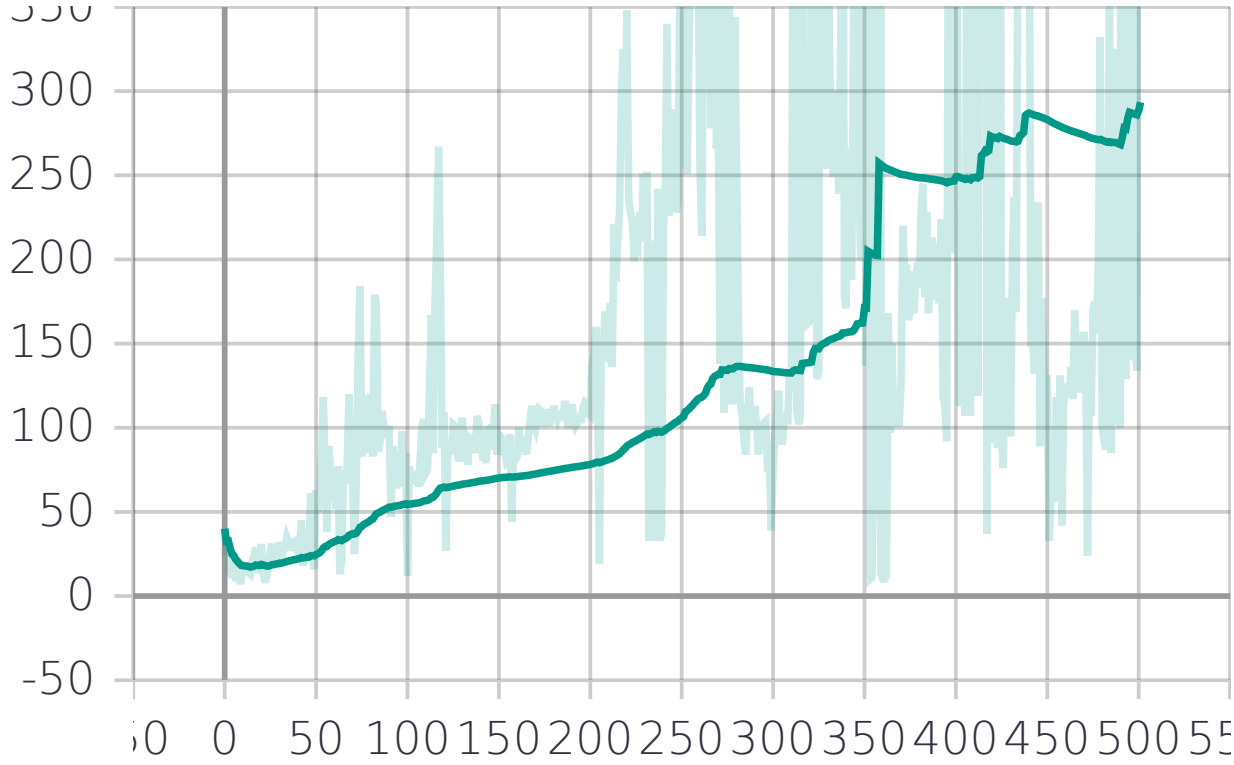
```
>>> from spikingjelly.clock_driven.examples import Spiking_DQN_state
>>> Spiking_DQN_state.play(use_cuda=False, pt_path='./model/CartPole-v0/policy_net_
↪256_max.pt', env_name='CartPole-v0', hidden_size=256, played_frames=300)
```

训练好的 SNN 会控制 CartPole 的左右移动，直到游戏结束或持续帧数超过 `played_frames`。`play` 函数中会画出 SNN 中 IF 神经元在仿真期间的脉冲发放频率，以及输出层 NonSpikingLIF 神经元在最后时刻的电压：

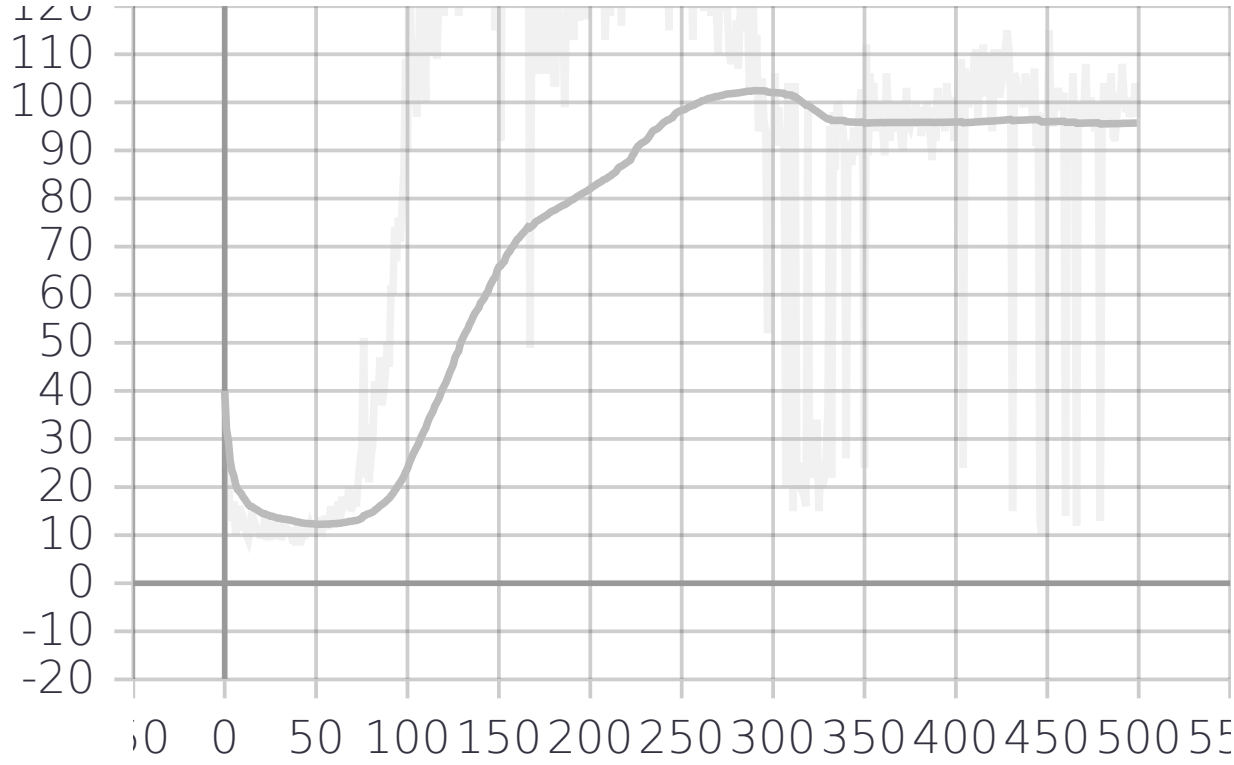
训练 16 次的效果：

训练 32 次的效果：

训练 500 个回合的性能曲线：



用相同处理方式的 ANN 训练 500 个回合的性能曲线 (完整的代码可见于 clock_driven/examples/DQN_state.py)：



1.9 强化学习 A2C

本教程作者：lucifer2859

本节教程使用 SNN 重新实现 actor-critic.py。请确保你已经阅读了原版代码以及相关论文，因为本教程是对原代码的扩展。

状态输入同 DQN 一样我们使用另一种常用的使 SNN 输出浮点值的方法：将神经元的阈值设置成无穷大，使其不发放脉冲，用神经元最后时刻的电压作为输出值。神经元实现这种神经元非常简单，只需要继承已有神经元，重写 `forward` 函数即可。LIF 神经元的电压不像 IF 神经元那样是简单的积分，因此我们使用 LIF 神经元来改写：

```
class NonSpikingLIFNode(neuron.LIFNode):
    def forward(self, dv: torch.Tensor):
        self.neuronal_charge(dv)
        # self.neuronal_fire()
        # self.neuronal_reset()
        return self.v
```

接下来，搭建我们的 Spiking Actor-Critic Network，网络的结构非常简单，全连接-IF 神经元-全连接-NonSpikingLIF 神经元，全连接-IF 神经元起到编码器的作用，而全连接-NonSpikingLIF 神经元则可以看作一个决策器：

```
class ActorCritic(nn.Module):
    def __init__(self, num_inputs, num_outputs, hidden_size, T=16):
        super(ActorCritic, self).__init__()

        self.critic = nn.Sequential(
            nn.Linear(num_inputs, hidden_size),
            neuron.IFNode(),
            nn.Linear(hidden_size, 1),
            NonSpikingLIFNode(tau=2.0)
        )

        self.actor = nn.Sequential(
            nn.Linear(num_inputs, hidden_size),
            neuron.IFNode(),
            nn.Linear(hidden_size, num_outputs),
            NonSpikingLIFNode(tau=2.0)
        )

        self.T = T

    def forward(self, x):
        for t in range(self.T):
```

(下页继续)

(续上页)

```
            self.critic(x)
            self.actor(x)
        value = self.critic[-1].v
        probs = F.softmax(self.actor[-1].v, dim=1)
        dist  = Categorical(probs)

        return dist, value
```

1.9.1 训练网络

训练部分的代码，与 ANN 版本几乎相同，使用 env 返回的 Observation 作为输入。

SNN 的训练代码如下，我们会保存训练过程中使得奖励最大的模型参数：

```
while step_idx < max_steps:

    log_probs = []
    values    = []
    rewards   = []
    masks     = []
    entropy = 0

    for _ in range(num_steps):
        state = torch.FloatTensor(state).to(device)
        dist, value = model(state)
        functional.reset_net(model)

        action = dist.sample()
        next_state, reward, done, _ = envs.step(action.cpu().numpy())

        log_prob = dist.log_prob(action)
        entropy += dist.entropy().mean()

        log_probs.append(log_prob)
        values.append(value)
        rewards.append(torch.FloatTensor(reward).unsqueeze(1).to(device))
        masks.append(torch.FloatTensor(1 - done).unsqueeze(1).to(device))

        state = next_state
        step_idx += 1

        if step_idx % 1000 == 0:
            test_reward = test_env()
```

(下页继续)

(续上页)

```
            print('Step: %d, Reward: %.2f' % (step_idx, test_reward))
            writer.add_scalar('Spiking-A2C-multi_env-' + env_name + '/Reward', test_
↪reward, step_idx)

    next_state = torch.FloatTensor(next_state).to(device)
    _, next_value = model(next_state)
    functional.reset_net(model)
    returns = compute_returns(next_value, rewards, masks)

    log_probs = torch.cat(log_probs)
    returns   = torch.cat(returns).detach()
    values    = torch.cat(values)

    advantage = returns - values

    actor_loss  = - (log_probs * advantage.detach()).mean()
    critic_loss = advantage.pow(2).mean()

    loss = actor_loss + 0.5 * critic_loss - 0.001 * entropy

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

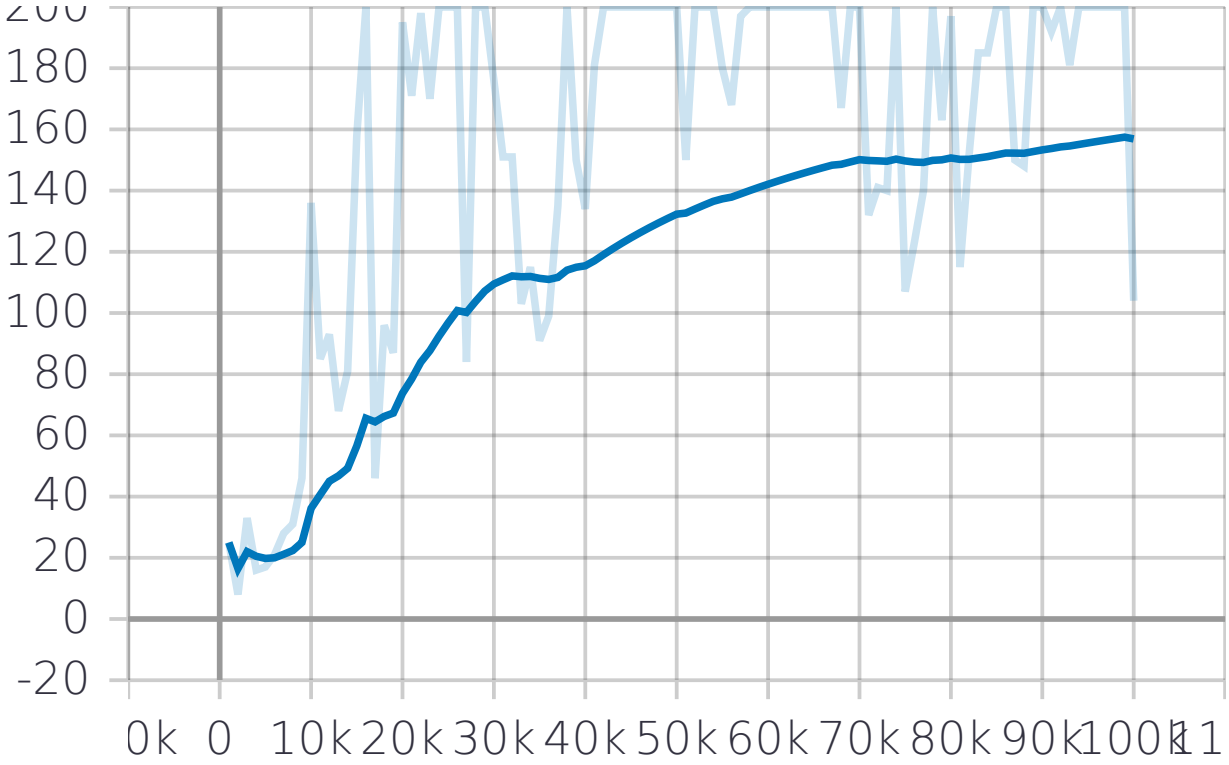
另外一个需要注意的地方是，SNN 是有状态的，因此每次前向传播后，不要忘了将网络 `reset`。

完整的代码可见于 clock_driven/examples/Spiking_A2C.py。可以从命令行直接启动训练：

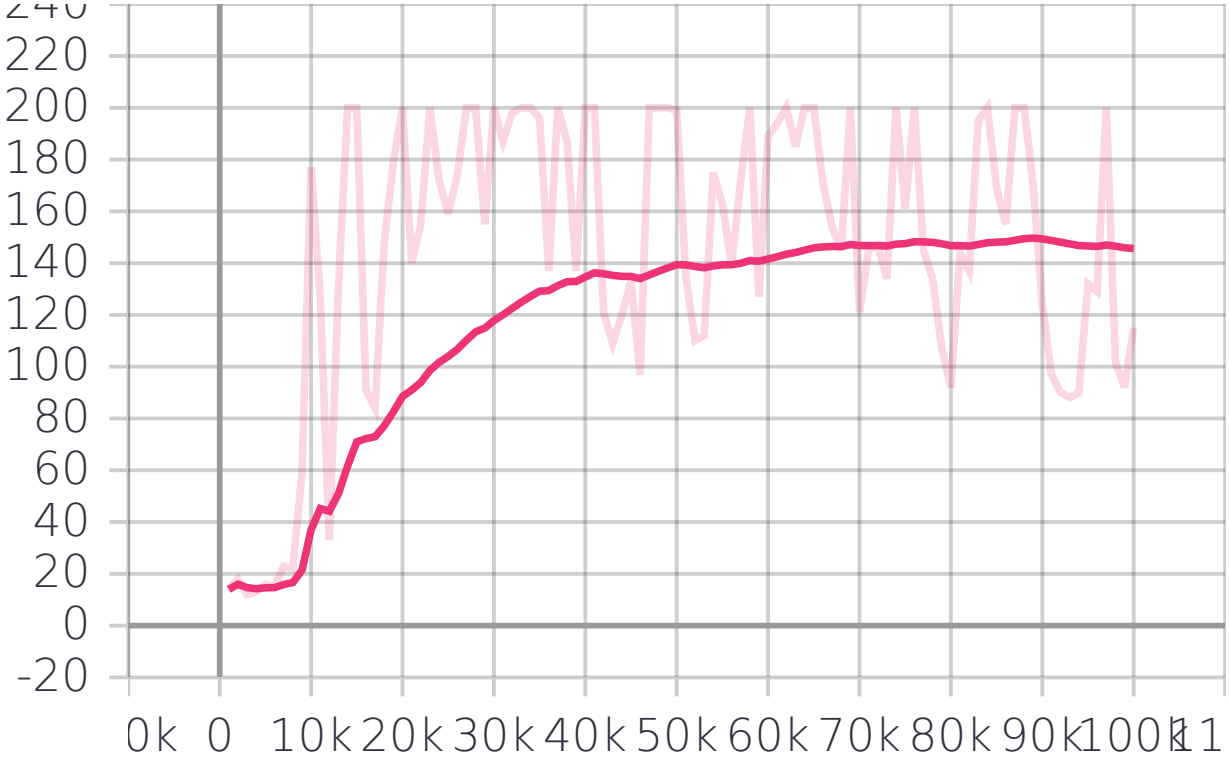
```
>>> python Spiking_A2C.py
```

1.9.2 ANN 与 SNN 的性能对比

训练 1e5 个步骤的性能曲线：



用相同处理方式的 ANN 训练 1e5 个步骤的性能曲线 (完整的代码可见于 clock_driven/examples/A2C.py)：



1.10 强化学习 PPO

本教程作者：lucifer2859

本节教程使用 SNN 重新实现 ppo.py。请确保你已经阅读了原版代码以及相关论文，因为本教程是对原代码的扩展。

状态输入同 DQN 一样我们使用另一种常用的使 SNN 输出浮点值的方法：将神经元的阈值设置成无穷大，使其不发放脉冲，用神经元最后时刻的电压作为输出值。神经元实现这种神经元非常简单，只需要继承已有神经元，重写 `forward` 函数即可。LIF 神经元的电压不像 IF 神经元那样是简单的积分，因此我们使用 LIF 神经元来改写：

```
class NonSpikingLIFNode(neuron.LIFNode):
    def forward(self, dv: torch.Tensor):
        self.neuronal_charge(dv)
        # self.neuronal_fire()
        # self.neuronal_reset()
        return self.v
```

接下来，搭建我们的 Spiking Actor-Critic Network，网络的结构非常简单，全连接-IF 神经元-全连接-NonSpikingLIF 神经元，全连接-IF 神经元起到编码器的作用，而全连接-NonSpikingLIF 神经元则可以看作一个决策器：

```
class ActorCritic(nn.Module):
    def __init__(self, num_inputs, num_outputs, hidden_size, T=16, std=0.0):
        super(ActorCritic, self).__init__()

        self.critic = nn.Sequential(
            nn.Linear(num_inputs, hidden_size),
            neuron.IFNode(),
            nn.Linear(hidden_size, 1),
            NonSpikingLIFNode(tau=2.0)
        )

        self.actor = nn.Sequential(
            nn.Linear(num_inputs, hidden_size),
            neuron.IFNode(),
            nn.Linear(hidden_size, num_outputs),
            NonSpikingLIFNode(tau=2.0)
        )

        self.log_std = nn.Parameter(torch.ones(1, num_outputs) * std)

        self.T = T
```

(下页继续)

(续上页)

```
    def forward(self, x):
        for t in range(self.T):
            self.critic(x)
            self.actor(x)
        value = self.critic[-1].v
        mu    = self.actor[-1].v
        std   = self.log_std.exp().expand_as(mu)
        dist  = Normal(mu, std)
        return dist, value
```

1.10.1 训练网络

训练部分的代码，与 ANN 版本几乎相同，使用 env 返回的 Observation 作为输入。

SNN 的训练代码如下，我们会保存训练过程中使得奖励最大的模型参数：

```
# GAE
def compute_gae(next_value, rewards, masks, values, gamma=0.99, tau=0.95):
    values = values + [next_value]
    gae = 0
    returns = []
    for step in reversed(range(len(rewards))):
        delta = rewards[step] + gamma * values[step + 1] * masks[step] - values[step]
        gae = delta + gamma * tau * masks[step] * gae
        returns.insert(0, gae + values[step])
    return returns

# Proximal Policy Optimization Algorithm
# Arxiv: "https://arxiv.org/abs/1707.06347"
def ppo_iter(mini_batch_size, states, actions, log_probs, returns, advantage):
    batch_size = states.size(0)
    ids = np.random.permutation(batch_size)
    ids = np.split(ids[:batch_size // mini_batch_size * mini_batch_size], batch_size /
↪/ mini_batch_size)
    for i in range(len(ids)):
        yield states[ids[i], :], actions[ids[i], :], log_probs[ids[i], :],␣
↪returns[ids[i], :], advantage[ids[i], :]

def ppo_update(ppo_epochs, mini_batch_size, states, actions, log_probs, returns,␣
↪advantages, clip_param=0.2):
    for _ in range(ppo_epochs):
        for state, action, old_log_probs, return_, advantage in ppo_iter(mini_batch_
↪size, states, actions, log_probs, returns, advantages):
```

(下页继续)

(续上页)

```
            dist, value = model(state)
            functional.reset_net(model)
            entropy = dist.entropy().mean()
            new_log_probs = dist.log_prob(action)

            ratio = (new_log_probs - old_log_probs).exp()
            surr1 = ratio * advantage
            surr2 = torch.clamp(ratio, 1.0 - clip_param, 1.0 + clip_param) * advantage

            actor_loss  = - torch.min(surr1, surr2).mean()
            critic_loss = (return_ - value).pow(2).mean()

            loss = 0.5 * critic_loss + actor_loss - 0.001 * entropy

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

while step_idx < max_steps:

    log_probs = []
    values    = []
    states    = []
    actions   = []
    rewards   = []
    masks     = []
    entropy = 0

    for _ in range(num_steps):
        state = torch.FloatTensor(state).to(device)
        dist, value = model(state)
        functional.reset_net(model)

        action = dist.sample()
        next_state, reward, done, _ = envs.step(torch.max(action, 1)[1].cpu().numpy())

        log_prob = dist.log_prob(action)
        entropy += dist.entropy().mean()

        log_probs.append(log_prob)
        values.append(value)
        rewards.append(torch.FloatTensor(reward).unsqueeze(1).to(device))
        masks.append(torch.FloatTensor(1 - done).unsqueeze(1).to(device))
```

(下页继续)

(续上页)

```
        states.append(state)
        actions.append(action)

        state = next_state
        step_idx += 1

        if step_idx % 100 == 0:
            test_reward = test_env()
            print('Step: %d, Reward: %.2f' % (step_idx, test_reward))
            writer.add_scalar('Spiking-PPO-' + env_name + '/Reward', test_reward,␣
↪step_idx)

    next_state = torch.FloatTensor(next_state).to(device)
    _, next_value = model(next_state)
    functional.reset_net(model)
    returns = compute_gae(next_value, rewards, masks, values)

    returns   = torch.cat(returns).detach()
    log_probs = torch.cat(log_probs).detach()
    values    = torch.cat(values).detach()
    states    = torch.cat(states)
    actions   = torch.cat(actions)
    advantage = returns - values

    ppo_update(ppo_epochs, mini_batch_size, states, actions, log_probs, returns,␣
↪advantage)
```

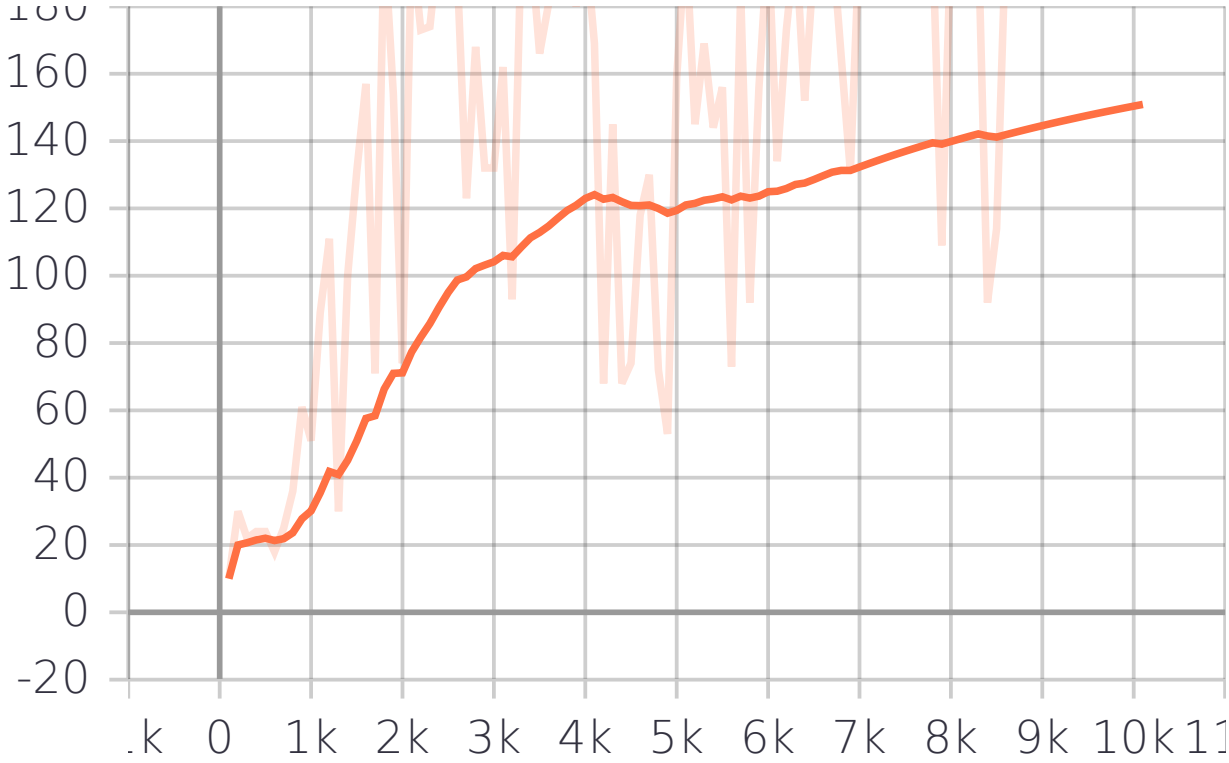
另外一个需要注意的地方是，SNN 是有状态的，因此每次前向传播后，不要忘了将网络 `reset`。

完整的代码可见于 clock_driven/examples/Spiking_PPO.py。可以从命令行直接启动训练：

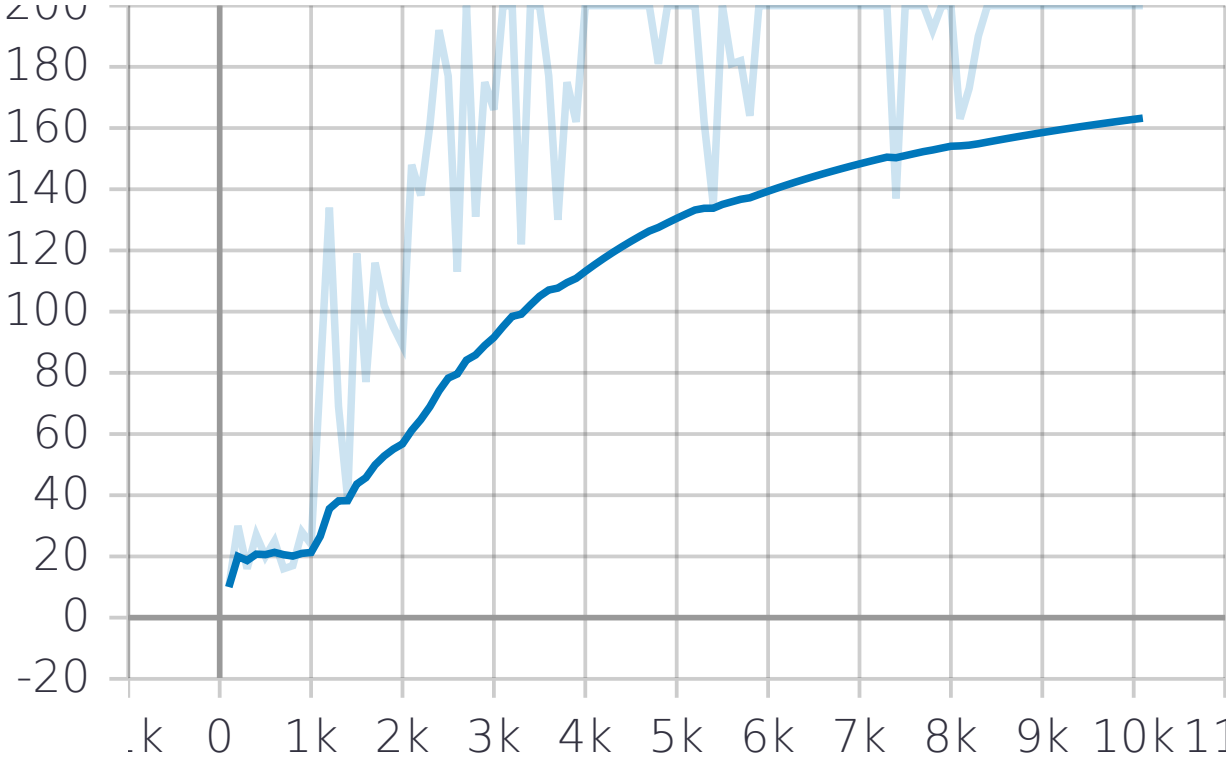
```
>>> python Spiking_PPO.py
```

1.10.2 ANN 与 SNN 的性能对比

训练 1e5 个步骤的性能曲线：



用相同处理方式的 ANN 训练 1e5 个步骤的性能曲线 (完整的代码可见于 clock_driven/examples/PPO.py)：



1.11 利用 Spiking LSTM 实现基于文本的姓氏分类任务

本教程作者：LiutaoYu，fangwei123456

本节教程使用 Spiking LSTM 重新实现 PyTorch 的官方教程 NLP From Scratch: Classifying Names with a Character-Level RNN。对应的中文版教程可参见 使用字符级别特征的 RNN 网络进行名字分类。请确保你已经阅读了原版教程和代码，因为本教程是对原教程的扩展。本教程将构建和训练字符级的 Spiking LSTM 来对姓氏进行分类。具体而言，本教程将在 18 种语言构成的几千个姓氏的数据集上训练 Spiking LSTM 模型，网络可根据一个姓氏的拼写预测其属于哪种语言。完整代码可见于 clock_driven/examples/spiking_lstm_text.py。

1.11.1 准备数据

首先，我们参照原教程下载数据，并进行预处理。预处理后，我们可以得到一个语言对应姓氏列表的字典，即 `{language: [names ...]}` 。进一步地，我们将数据集按照 4:1 的比例划分为训练集和测试集，即 `category_lines_train` 和 `category_lines_test` 。这里还需要留意几个后续会经常使用的变量：`all_categories` 是全部语言种类的列表，`n_categories=18` 则是语言种类的数量，`n_letters=58` 是组成 `names` 的所有字母和符号的集合的元素数量。

```
# split the data into training set and testing set
numExamplesPerCategory = []
category_lines_train = {}
category_lines_test = {}
testNumtot = 0
for c, names in category_lines.items():
    category_lines_train[c] = names[:int(len(names)*0.8)]
    category_lines_test[c] = names[int(len(names)*0.8):]
    numExamplesPerCategory.append([len(category_lines[c]), len(category_lines_
↪train[c]), len(category_lines_test[c])])
    testNumtot += len(category_lines_test[c])
```

此外，我们改写了原教程中的 `randomTrainingExample()` 函数，以便在不同条件下进行使用。注意此处利用了原教程中定义的 `lineToTensor()` 和 `randomChoice()` 两个函数。前者用于将单词转化为 one-hot 张量，后者用于从数据集中随机抽取一个样本。

```
# Preparing [x, y] pair
def randomPair(sampleSource):
    """
    Args:
        sampleSource:  'train', 'test', 'all'
    Returns:
        category, line, category_tensor, line_tensor
    """
    category = randomChoice(all_categories)
```

(下页继续)

(续上页)

```
    if sampleSource == 'train':
        line = randomChoice(category_lines_train[category])
    elif sampleSource == 'test':
        line = randomChoice(category_lines_test[category])
    elif sampleSource == 'all':
        line = randomChoice(category_lines[category])
    category_tensor = torch.tensor([all_categories.index(category)], dtype=torch.
↪float)
    line_tensor = lineToTensor(line)
    return category, line, category_tensor, line_tensor
```

1.11.2 构造 Spiking LSTM 神经网络

我们利用 spikingjelly 中的 rnn 模块（`rnn.SpikingLSTM()`）来搭建 Spiking LSTM 神经网络。其工作原理可参见论文 Long Short-Term Memory Spiking Networks and Their Applications 。输入层神经元个数等于 `n_letters` ，隐藏层神经元个数 `n_hidden` 可自行定义，输出层神经元个数等于 `n_categories` 。我们在 LSTM 的输出层之后接一个全连接层，并利用 softmax 函数对全连接层的数据进行处理以获取类别概率。

```
from spikingjelly.clock_driven import rnn
n_hidden = 256

class Net(nn.Module):
    def __init__(self, n_letters, n_hidden, n_categories):
        super().__init__()
        self.n_input = n_letters
        self.n_hidden = n_hidden
        self.n_out = n_categories
        self.lstm = rnn.SpikingLSTM(self.n_input, self.n_hidden, 1)
        self.fc = nn.Linear(self.n_hidden, self.n_out)

    def forward(self, x):
        x, _ = self.lstm(x)
        output = self.fc(x[-1])
        output = F.softmax(output, dim=1)
        return output
```

1.11.3 网络训练

首先，我们初始化网络 net ，并定义训练时长 TRAIN_EPISODES 、学习率 learning_rate 等。这里我们采用 mse_loss 损失函数和 Adam 优化器来对训练网络。单个 epoch 的训练流程大致如下：1）从训练集中随机抽取一个样本，获取输入和标签，并转化为 tensor；2）网络接收输入，进行前向过程，获取各类别的预测概率；3）利用 mse_loss 函数获取网络预测概率和真实标签 one-hot 编码之间的差距，即网络损失；4）梯度反传，并更新网络参数；5）判断此次预测是否正确，并累计预测正确的数量，以获取模型在训练过程中针对训练集的准确率（每隔 plot_every 个 epoch 计算一次）；6）每隔 plot_every 个 epoch 在测试集上测试一次，并统计测试准确率。此外，在训练过程中，我们会记录网络损失 avg_losses 、训练集准确率 accuracy_rec 和测试集准确率 test_accu_rec ，以便于观察训练效果，并在训练完成后绘制图片。在训练完成之后，我们会保存网络的最终状态以用于测试；同时，也可以保存相关变量，以便后续分析。

```
# IF_TRAIN = 1
TRAIN_EPISODES = 1000000
plot_every = 1000
learning_rate = 1e-4

net = Net(n_letters, n_hidden, n_categories)
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)

print('Training...')
current_loss = 0
correct_num = 0
avg_losses = []
accuracy_rec = []
test_accu_rec = []
start = time.time()
for epoch in range(1, TRAIN_EPISODES+1):
    net.train()
    category, line, category_tensor, line_tensor = randomPair('train')
    label_one_hot = F.one_hot(category_tensor.to(int), n_categories).float()

    optimizer.zero_grad()
    out_prob_log = net(line_tensor)
    loss = F.mse_loss(out_prob_log, label_one_hot)
    loss.backward()
    optimizer.step()

    # 优化一次参数后，需要重置网络的状态。是否需要？结果差别不明显！(2020.11.3)
    # functional.reset_net(net)

    current_loss += loss.data.item()

    guess, _ = categoryFromOutput(out_prob_log.data)
```

(下页继续)

(续上页)

```
    if guess == category:
        correct_num += 1

    # Add current loss avg to list of losses
    if epoch % plot_every == 0:
        avg_losses.append(current_loss / plot_every)
        accuracy_rec.append(correct_num / plot_every)
        current_loss = 0
        correct_num = 0

    # 每训练一定次数即进行一次测试
    if epoch % plot_every == 0:  # int(TRAIN_EPISODES/1000)
        net.eval()
        with torch.no_grad():
            numCorrect = 0
            for i in range(n_categories):
                category = all_categories[i]
                for tname in category_lines_test[category]:
                    output = net(lineToTensor(tname))
                    # 运行一次后，需要重置网络的状态。是否需要？
                    # functional.reset_net(net)
                    guess, _ = categoryFromOutput(output.data)
                    if guess == category:
                        numCorrect += 1
            test_accu = numCorrect / testNumtot
            test_accu_rec.append(test_accu)
            print('Epoch %d %d%% (%s); Avg_loss %.4f; Train accuracy %.4f; Test␣
↪accuracy %.4f' % (
                epoch, epoch / TRAIN_EPISODES * 100, timeSince(start), avg_losses[-1],
↪ accuracy_rec[-1], test_accu))

torch.save(net, 'char_rnn_classification.pth')
np.save('avg_losses.npy', np.array(avg_losses))
np.save('accuracy_rec.npy', np.array(accuracy_rec))
np.save('test_accu_rec.npy', np.array(test_accu_rec))
np.save('category_lines_train.npy', category_lines_train, allow_pickle=True)
np.save('category_lines_test.npy', category_lines_test, allow_pickle=True)
# x = np.load('category_lines_test.npy', allow_pickle=True)  # 读取数据的方法
# xdict = x.item()

plt.figure()
plt.subplot(311)
plt.plot(avg_losses)
```

(下页继续)

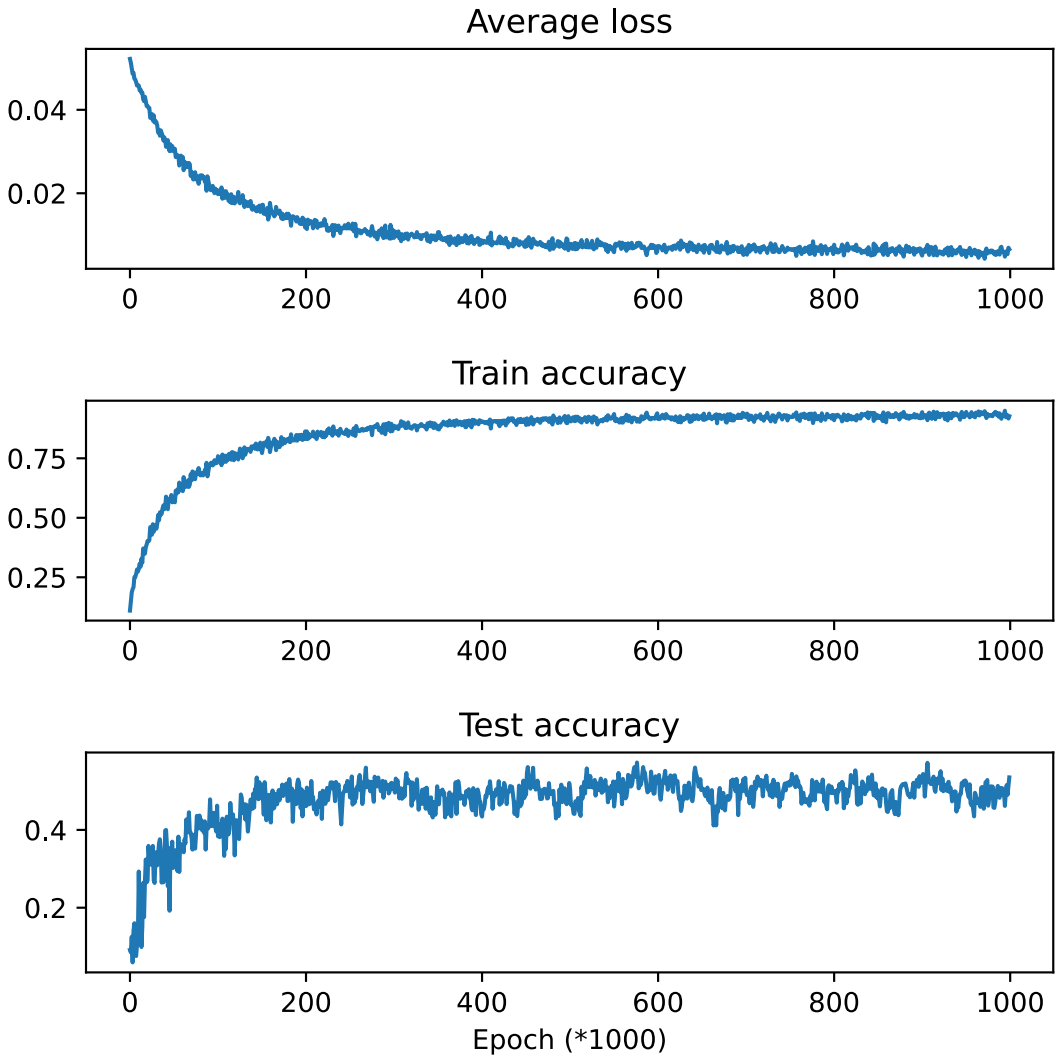
(续上页)

```
plt.title('Average loss')
plt.subplot(312)
plt.plot(accuracy_rec)
plt.title('Train accuracy')
plt.subplot(313)
plt.plot(test_accu_rec)
plt.title('Test accuracy')
plt.xlabel('Epoch (*1000)')
plt.subplots_adjust(hspace=0.6)
plt.savefig('TrainingProcess.svg')
plt.close()
```

设定 `IF_TRAIN = 1`，在 Python Console 中运行 `%run ./spiking_lstm_text.py`，输出如下：

```
Backend Qt5Agg is interactive backend. Turning interactive mode on.
Training...
Epoch 1000 0% (0m 18s); Avg_loss 0.0525; Train accuracy 0.0830; Test accuracy 0.0806
Epoch 2000 0% (0m 37s); Avg_loss 0.0514; Train accuracy 0.1470; Test accuracy 0.1930
Epoch 3000 0% (0m 55s); Avg_loss 0.0503; Train accuracy 0.1650; Test accuracy 0.0537
Epoch 4000 0% (1m 14s); Avg_loss 0.0494; Train accuracy 0.1920; Test accuracy 0.0938
...
...
Epoch 998000 99% (318m 54s); Avg_loss 0.0063; Train accuracy 0.9300; Test accuracy 0.
↪5036
Epoch 999000 99% (319m 14s); Avg_loss 0.0056; Train accuracy 0.9380; Test accuracy 0.
↪5004
Epoch 1000000 100% (319m 33s); Avg_loss 0.0055; Train accuracy 0.9340; Test accuracy␣
↪0.5118
```

下图展示了训练过程中损失函数、测试集准确率、测试集准确率随时间的变化。值得注意的一点是，测试表明，在当前 Spiking LSTM 网络中是否在一次运行完成后重置网络 `functional.reset_net(net)` 对于结果没有显著的影响。我们猜测是因为当前网络输入是随时间变化的，而且网络自身需要运行一段时间后才会输出分类结果，因此网络初始状态影响不显著。



1.11.4 网络测试

在测试过程中，我们首先需要导入训练完成后存储的网络，随后进行三方面的测试：（1）计算最终的测试集准确率；（2）让用户输入姓氏拼写以预测其属于哪种语言；（3）计算 Confusion matrix，每一行表示当样本源于某一个类别时，网络预测其属于各类别的概率，即对角线表示预测正确的概率。

```
# IF_TRAIN = 0
print('Testing...')

net = torch.load('char_rnn_classification.pth')
```

（下页继续）

(续上页)

```
# 遍历测试集计算准确率
print('Calculating testing accuracy...')
numCorrect = 0
for i in range(n_categories):
    category = all_categories[i]
    for tname in category_lines_test[category]:
        output = net(lineToTensor(tname))
        # 运行一次后，需要重置网络的状态。是否需要？
        # functional.reset_net(net)
        guess, _ = categoryFromOutput(output.data)
        if guess == category:
            numCorrect += 1
test_accu = numCorrect / testNumtot
print('Test accuracy: {:.3f}, Random guess: {:.3f}'.format(test_accu, 1/n_categories))

# 让用户输入姓氏以判断其属于哪种语系
n_predictions = 3
for j in range(3):
    first_name = input('请输入一个姓氏以判断其属于哪种语系: ')
    print('\n> %s' % first_name)
    output = net(lineToTensor(first_name))
    # 运行一次后，需要重置网络的状态。是否需要？
    # functional.reset_net(net)
    # Get top N categories
    topv, topi = output.topk(n_predictions, 1, True)
    predictions = []

    for i in range(n_predictions):
        value = topv[0][i].item()
        category_index = topi[0][i].item()
        print('(%.2f) %s' % (value, all_categories[category_index]))
        predictions.append([value, all_categories[category_index]])

# 计算 confusion 矩阵
print('Calculating confusion matrix...')
confusion = torch.zeros(n_categories, n_categories)
n_confusion = 10000

# Keep track of correct guesses in a confusion matrix
for i in range(n_confusion):
    category, line, category_tensor, line_tensor = randomPair('all')
    output = net(line_tensor)
    # 运行一次后，需要重置网络的状态。是否需要？
```

(下页继续)

(续上页)

```
    # functional.reset_net(net)
    guess, guess_i = categoryFromOutput(output.data)
    category_i = all_categories.index(category)
    confusion[category_i][guess_i] += 1

confusion = confusion / confusion.sum(1)
np.save('confusion.npy', confusion)

# Set up plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111)
cax = ax.matshow(confusion.numpy())
fig.colorbar(cax)
# Set up axes
ax.set_xticklabels(['] + all_categories, rotation=90)
ax.set_yticklabels(['] + all_categories)
# Force label at every tick
ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
ax.yaxis.set_major_locator(ticker.MultipleLocator(1))
# sphinx_gallery_thumbnail_number = 2
plt.show()
plt.savefig('ConfusionMatrix.svg')
plt.close()
```

设定 `IF_TRAIN = 0`，在 Python Console 中运行 `%run ./spiking_lstm_text.py`，输出如下：

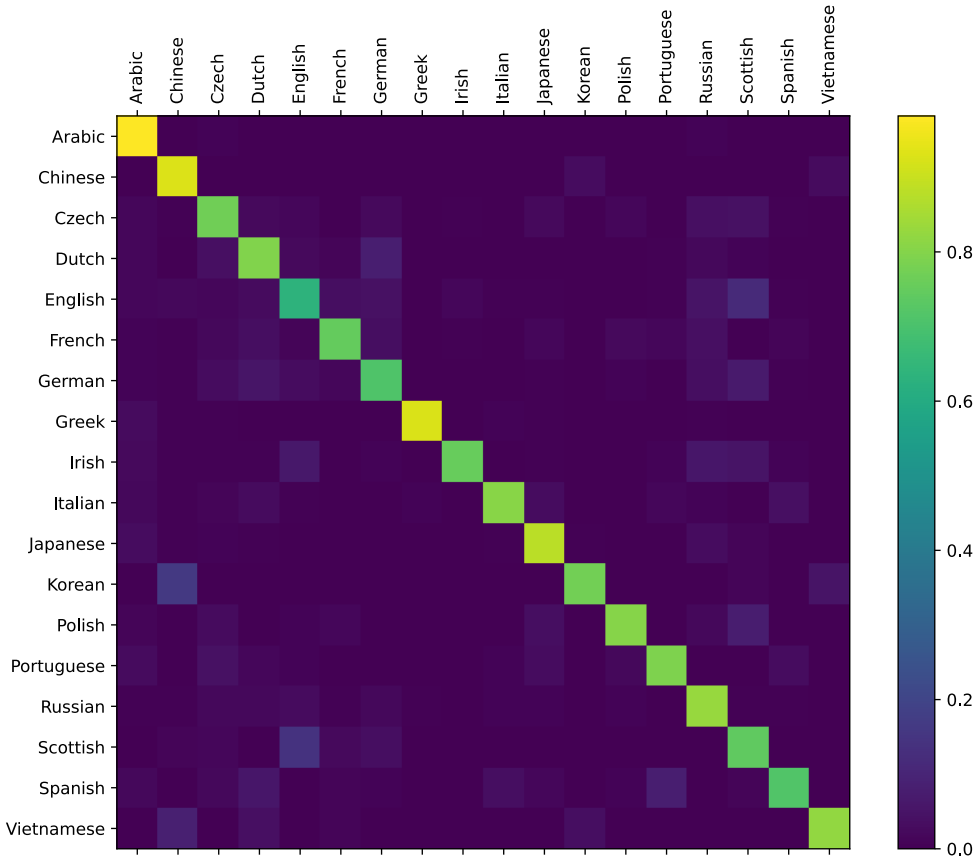
```
Testing...
Calculating testing accuracy...
Test accuracy: 0.512, Random guess: 0.056
请输入一个姓氏以判断其属于哪种语系:> YU
> YU
(0.18) Scottish
(0.12) English
(0.11) Italian
请输入一个姓氏以判断其属于哪种语系:> Yu
> Yu
(0.63) Chinese
(0.23) Korean
(0.07) Vietnamese
请输入一个姓氏以判断其属于哪种语系:> Zou
> Zou
(1.00) Chinese
(0.00) Arabic
(0.00) Polish
```

(下页继续)

(续上页)

```
Calculating confusion matrix...
```

下图展示了 Confusion matrix。对角线越亮，表示模型对某一类别预测最好，很少产生混淆，如 Arabic 和 Greek。而有的语言则较容易产生混淆，如 Korean 和 Chinese，Spanish 和 Portuguese，English 和 Scottish。



1.12 传播模式

本教程作者：fangwei123456

1.12.1 单步传播与多步传播

SpikingJelly 中的绝大多数模块（*spikingjelly.clock_driven.rnn* 除外），例如*spikingjelly.clock_driven.layer.Dropout*，模块名的前缀中没有 MultiStep，表示这个模块的 forward 函数定义的是单步的前向传播：

输入 X_t，输出 Y_t

而如果前缀中含有 MultiStep，例如*spikingjelly.clock_driven.layer.MultiStepDropout*，则表面这个模块的 forward 函数定义的是多步的前向传播：

输入 $X_t, t = 0, 1, ..., T-1$，输出 $Y_t, t = 0, 1, ..., T-1$

一个单步传播的模块，可以很容易被封装成多步传播的模块，*spikingjelly.clock_driven.layer.MultiStepContainer* 提供了非常简单的方式，将原始模块作为子模块，并在 forward 函数中实现在时间上的循环，代码如下所示：

```
class MultiStepContainer(nn.Module):
    def __init__(self, module: nn.Module):
        super().__init__()
        self.module = module

    def forward(self, x_seq: torch.Tensor):
        y_seq = []
        for t in range(x_seq.shape[0]):
            y_seq.append(self.module(x_seq[t]))
            y_seq[-1].unsqueeze_(0)
        return torch.cat(y_seq, 0)

    def reset(self):
        if hasattr(self.module, 'reset'):
            self.module.reset()
```

我们使用这种方式来包装一个 IF 神经元：

```
from spikingjelly.clock_driven import neuron, layer
import torch

neuron_num = 4
T = 8
if_node = neuron.IFNode()
x = torch.rand([T, neuron_num]) * 2
for t in range(T):
    print(f'if_node output spikes at t={t}', if_node(x[t]))
if_node.reset()
```

(下页继续)

(续上页)

```
ms_if_node = layer.MultiStepContainer(if_node)
print("multi step if_node output spikes\n", ms_if_node(x))
ms_if_node.reset()
```

输出为：

```
if_node output spikes at t=0 tensor([1., 1., 1., 0.])
if_node output spikes at t=1 tensor([0., 0., 0., 1.])
if_node output spikes at t=2 tensor([1., 1., 1., 1.])
if_node output spikes at t=3 tensor([0., 0., 1., 0.])
if_node output spikes at t=4 tensor([1., 1., 1., 1.])
if_node output spikes at t=5 tensor([1., 0., 0., 0.])
if_node output spikes at t=6 tensor([1., 0., 1., 1.])
if_node output spikes at t=7 tensor([1., 1., 1., 0.])
multi step if_node output spikes
 tensor([[1., 1., 1., 0.],
        [0., 0., 0., 1.],
        [1., 1., 1., 1.],
        [0., 0., 1., 0.],
        [1., 1., 1., 1.],
        [1., 0., 0., 0.],
        [1., 0., 1., 1.],
        [1., 1., 1., 0.]])
```

两种方式的输出是完全相同的。

1.12.2 逐步传播与逐层传播

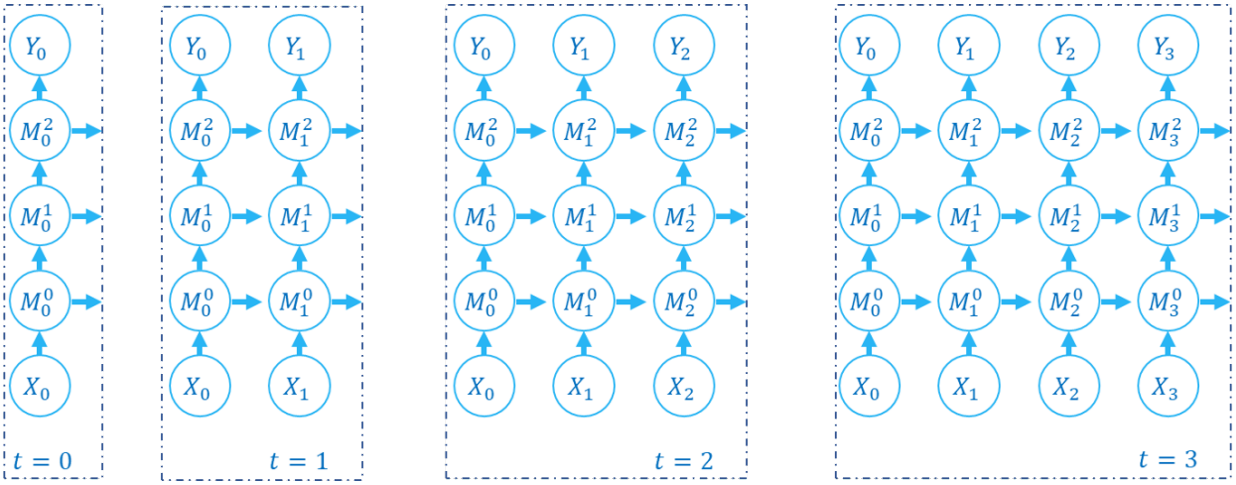
在以往的教程和样例中，我们定义的网络在运行时，是按照 *逐步传播 (step-by-step)* 的方式，例如上文中的：

```
if_node = neuron.IFNode()
x = torch.rand([T, neuron_num]) * 2
for t in range(T):
    print(f'if_node output spikes at t={t}', if_node(x[t]))
```

逐步传播 (step-by-step)，指的是在前向传播时，先计算出整个网络在 $t=0$ 的输出 Y_0，然后再计算整个网络在 $t=1$ 的输出 Y_1，……，最终得到网络在所有时刻的输出 $Y_t, t=0,1,...,T-1$。例如下面这份代码（假定 `M0`，`M1`，`M2` 都是单步传播的模块）：

```
net = nn.Sequential(M0, M1, M2)

for t in range(T):
    Y[t] = net(X[t])
```

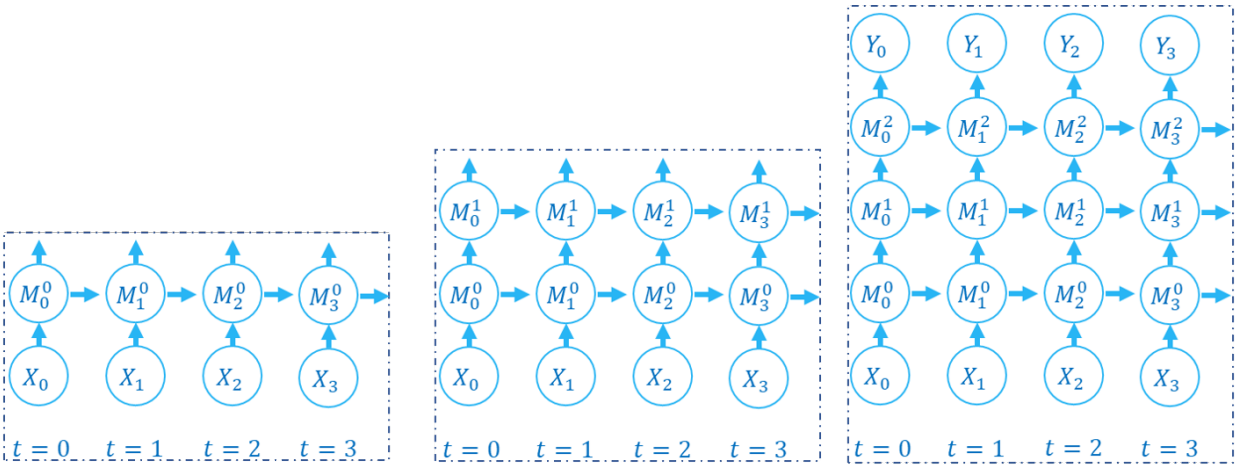
前向传播的计算图的构建顺序如下所示：



对于 SNN 以及 RNN，前向传播既发生在空域也发生在时域，逐步传播逐步计算出整个网络在不同时刻的状态，我们可以很容易联想到，还可以使用另一种顺序来计算：逐层计算出每一层网络在所有时刻的状态。例如下面这份代码（假定 `M0`，`M1`，`M2` 都是多步传播的模块）：

```
net = nn.Sequential(M0, M1, M2)

Y = net(X)
```

前向传播的计算图的构建顺序如下所示：



我们称这种方式为 *逐层传播 (layer-by-layer)*。逐层传播在 RNN 以及 SNN 中也被广泛使用，例如 Low-activity supervised convolutional spiking neural networks applied to speech commands recognition 通过逐层计算的方式来获取每一层在所有时刻的输出，然后在时域上进行卷积，代码可见于 https://github.com/romainzimmer/s2net。

逐步传播与 逐层传播遍历计算图的顺序不同，但计算的结果是完全相同的。但 逐层传播具有更大的并行性，因为当某一层是无状态的层，例如 `torch.nn.Linear`，逐步传播会按照下述方式计算：

```
for t in range(T):
    y[t] = fc(x[t])  # x.shape=[T, batch_size, in_features]
```

而逐层传播则可以并行计算：

```
y = fc(x)  # x.shape=[T, batch_size, in_features]
```

对于无状态的层，我们可以将 shape=[T, batch_size, ...] 的输入拼接成 shape=[T * batch_size, ...] 后，再送入这一层计算，避免在时间上的循环。*spikingjelly.clock_driven.layer.SeqToANNContainer* 在 forward 函数中进行了这样的实现。我们可以直接使用这个模块：

```
with torch.no_grad():
    T = 16
    batch_size = 8
    x = torch.rand([T, batch_size, 4])
    fc = SeqToANNContainer(nn.Linear(4, 2), nn.Linear(2, 3))
    print(fc(x).shape)
```

输出为：

```
torch.Size([16, 8, 3])
```

输出仍然满足 shape=[T, batch_size, ...]，可以直接送入到下一层网络。

1.13 使用 CUDA 增强的神经元与逐层传播进行加速

本教程作者：fangwei123456

1.13.1 CUDA 加速的神经元

spikingjelly.cext.neuron 中的神经元与 spikingjelly.clock_driven.neuron 中的同名神经元，在前向传播和反向传播时的计算结果完全相同。但 spikingjelly.cext.neuron 将各种运算都封装到了一个 CUDA 内核；spikingjelly.clock_driven.neuron 则是使用 PyTorch 来实现神经元，每一个 Python 函数都需要调用一次相应的 CUDA 后端，这种频繁的调用存在很大的开销。现在让我们通过一个简单的实验，来对比两个模块中 LIF 神经元的运行耗时：

```
from spikingjelly import cext
from spikingjelly.cext import neuron as cext_neuron
from spikingjelly.clock_driven import neuron, surrogate, layer
import torch
```

(下页继续)

(续上页)

```
def cal_forward_t(multi_step_neuron, x, repeat_times):
    with torch.no_grad():
        used_t = cext.cal_fun_t(repeat_times, x.device, multi_step_neuron, x)
        multi_step_neuron.reset()
        return used_t * 1000


def forward_backward(multi_step_neuron, x):
    multi_step_neuron(x).sum().backward()
    multi_step_neuron.reset()
    x.grad.zero_()


def cal_forward_backward_t(multi_step_neuron, x, repeat_times):
    x.requires_grad_(True)
    used_t = cext.cal_fun_t(repeat_times, x.device, forward_backward, multi_step_
↪neuron, x)
    return used_t * 1000


device = 'cuda:0'
lif = layer.MultiStepContainer(neuron.LIFNode(surrogate_function=surrogate.
↪ATan(alpha=2.0)))
lif_cuda = layer.MultiStepContainer(cext_neuron.LIFNode(surrogate_function='ATan',␣
↪alpha=2.0))
lif_cuda_tt = cext_neuron.MultiStepLIFNode(surrogate_function='ATan', alpha=2.0)
lif.to(device)
lif_cuda.to(device)
lif_cuda_tt.to(device)
N = 2 ** 20
print('forward')
lif.eval()
lif_cuda.eval()
lif_cuda_tt.eval()
for T in [8, 16, 32, 64, 128]:
    x = torch.rand(T, N, device=device)
    print(T, cal_forward_t(lif, x, 1024), cal_forward_t(lif_cuda, x, 1024), cal_
↪forward_t(lif_cuda_tt, x, 1024))

print('forward and backward')
lif.train()
lif_cuda.train()
lif_cuda_tt.train()
```

(下页继续)

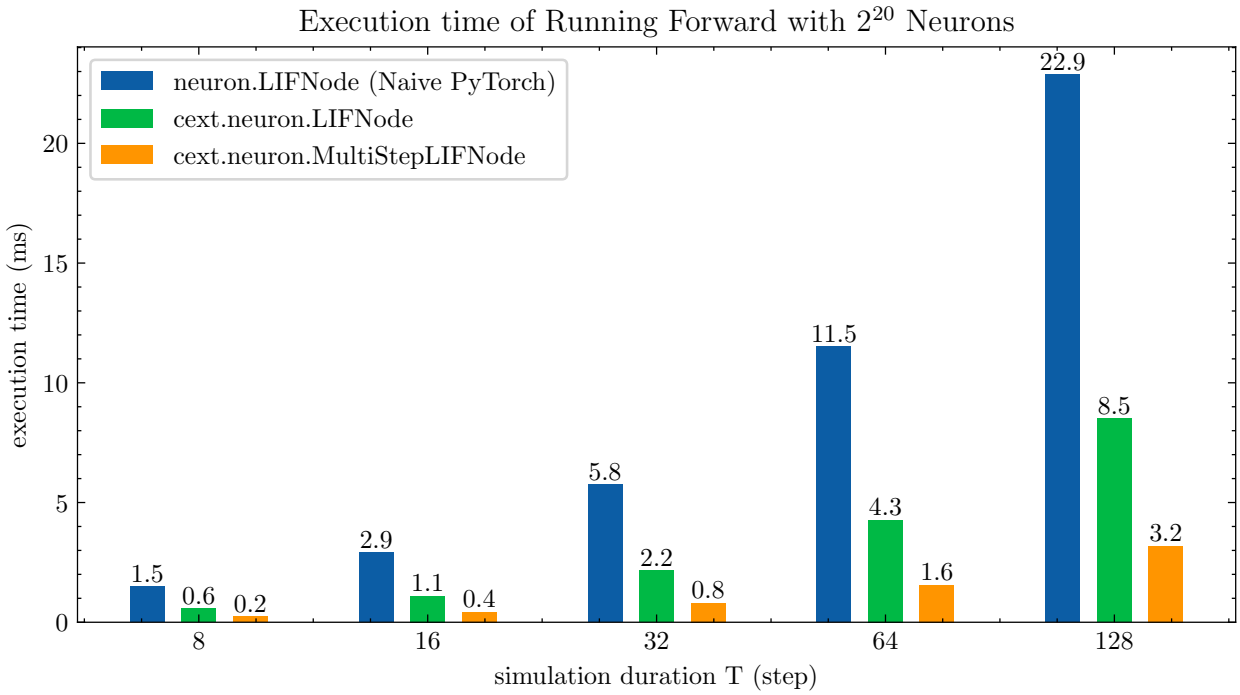
(续上页)

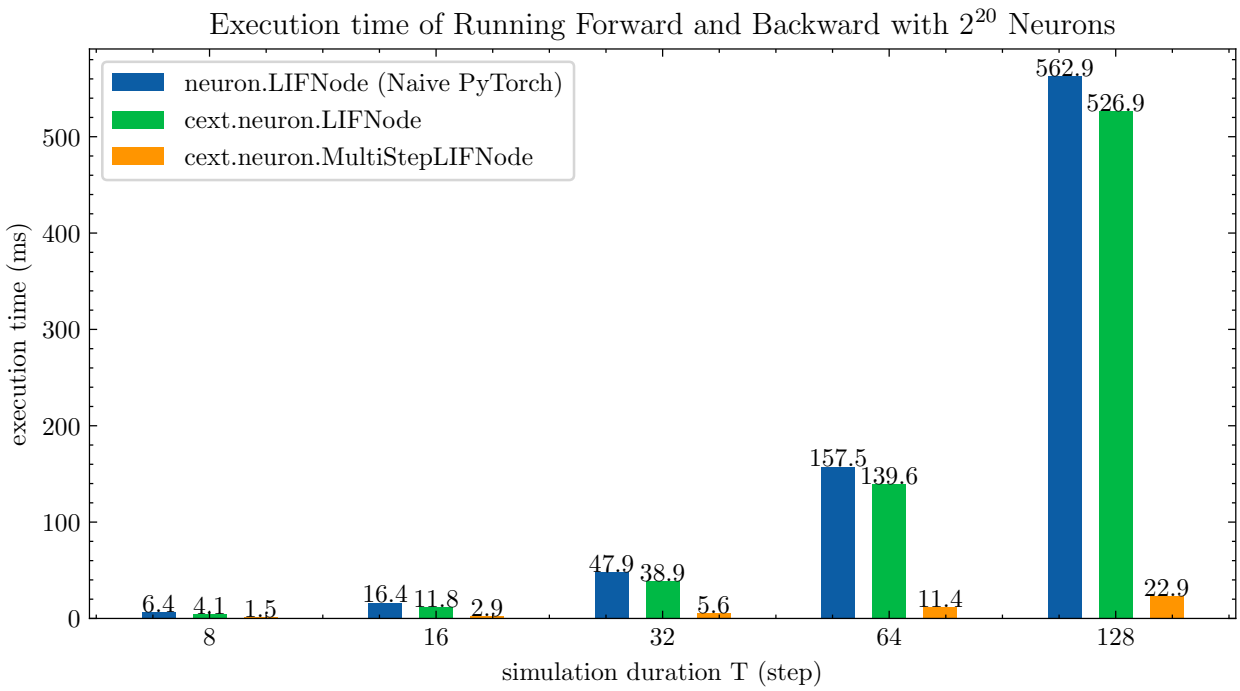
```
for T in [8, 16, 32, 64, 128]:
    x = torch.rand(T, N, device=device)
    print(T, cal_forward_backward_t(lif, x, 1024), cal_forward_backward_t(lif_cuda, x,
↪ 1024),
          cal_forward_backward_t(lif_cuda_tt, x, 1024))
```

实验机器使用 *Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz* 的 CPU 和 *GeForce RTX 2080 Ti* 的 GPU。运行结果如下：

```
forward
8 1.495931436011233 0.5668559867899603 0.23965466994013696
16 2.921243163427789 1.0935631946722424 0.42392046202621714
32 5.7503134660237265 2.1567279295595654 0.800143975766332
64 11.510705337741456 4.273202213653349 1.560730856454029
128 22.884282833274483 8.508097553431071 3.1778080651747587
forward and backward
8 6.444244811291355 4.052604411526772 1.4819166492543445
16 16.360167272978288 11.785220529191065 2.8625220465983148
32 47.86415797116206 38.88952818761027 5.645714411912195
64 157.53049964018828 139.59021832943108 11.367870506774125
128 562.9168437742464 526.8922436650882 22.945806705592986
```

将结果画成柱状图：





可以发现，使用 CUDA 封装操作的 `spikingjelly.cext.neuron` 速度明显快于原生 PyTorch 的神经元实现。

1.13.2 加速深度脉冲神经网络

现在让我们用 CUDA 封装的多步 LIF 神经元，重新实现*时间驱动：使用卷积 SNN 识别 Fashion-MNIST* 中的网络，并进行速度对比。我们只需要更改一下网络结构，无需进行其他的改动：

```
class Net(nn.Module):
    def __init__(self, tau, T, v_threshold=1.0, v_reset=0.0):
        super().__init__()
        self.T = T

        self.static_conv = nn.Sequential(
            nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(128),
        )

        self.conv = nn.Sequential(
            cext_neuron.MultiStepIFNode(v_threshold=v_threshold, v_reset=v_reset,␣
↪surrogate_function='ATan', alpha=2.0),
            layer.SeqToANNContainer(
                    nn.MaxPool2d(2, 2),  # 14 * 14
                    nn.Conv2d(128, 128, kernel_size=3, padding=1, bias=False),
                    nn.BatchNorm2d(128),
```

(下页继续)

(续上页)

```
            ),
            cext_neuron.MultiStepIFNode(v_threshold=v_threshold, v_reset=v_reset,␣
↪surrogate_function='ATan', alpha=2.0),
        )
        self.fc = nn.Sequential(
            layer.SeqToANNContainer(
                    nn.MaxPool2d(2, 2),  # 7 * 7
                    nn.Flatten(),
            ),
            layer.MultiStepDropout(0.5),
            layer.SeqToANNContainer(nn.Linear(128 * 7 * 7, 128 * 3 * 3, bias=False)),
            cext_neuron.MultiStepLIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_
↪reset, surrogate_function='ATan', alpha=2.0),
            layer.MultiStepDropout(0.5),
            layer.SeqToANNContainer(nn.Linear(128 * 3 * 3, 128, bias=False)),
            cext_neuron.MultiStepLIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_
↪reset, surrogate_function='ATan', alpha=2.0),
            layer.SeqToANNContainer(nn.Linear(128, 10, bias=False)),
            cext_neuron.MultiStepLIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_
↪reset, surrogate_function='ATan', alpha=2.0)
        )


    def forward(self, x):
        x_seq = self.static_conv(x).unsqueeze(0).repeat(self.T, 1, 1, 1, 1)
        # [N, C, H, W] -> [1, N, C, H, W] -> [T, N, C, H, W]

        out_spikes_counter = self.fc(self.conv(x_seq)).sum(0)
        return out_spikes_counter / self.T
```

完整的代码可见于 `spikingjelly.clock_driven.examples.conv_fashion_mnist_cuda_lbl`。我们按照与*时间驱动：使用卷积 SNN 识别 Fashion-MNIST* 中完全相同的输入参数和设备（*Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz* 的 CPU 和 *GeForce RTX 2080 Ti* 的 GPU）来运行，结果如下：

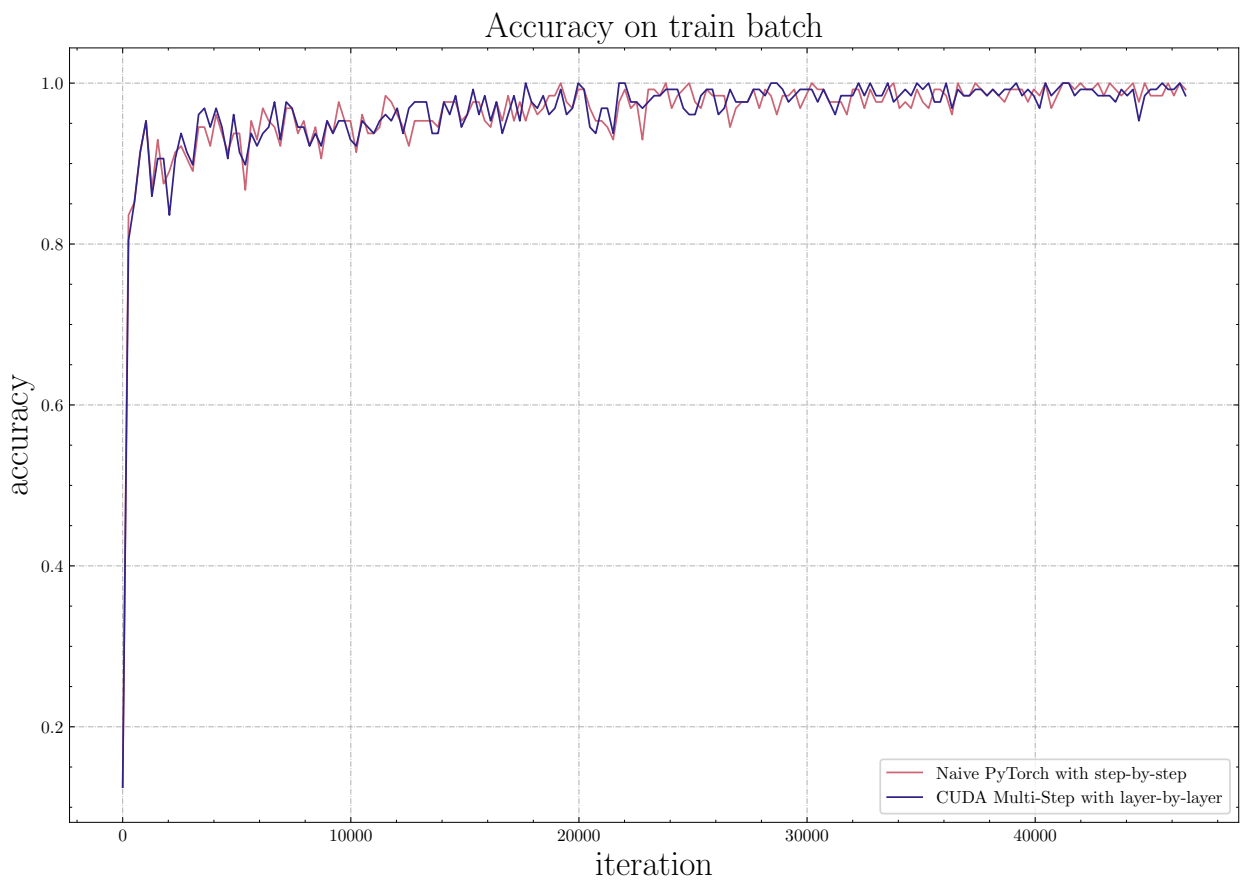
```
saving net...
saved
epoch=0, t_train=26.745780434459448, t_test=1.4819979975000024, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2,␣
↪max_test_accuracy=0.8705, train_times=468
saving net...
saved
epoch=1, t_train=26.087690989486873, t_test=1.502928489819169, device=cuda:0, dataset_
↪dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2, max_test_
↪accuracy=0.8913, train_times=936
```

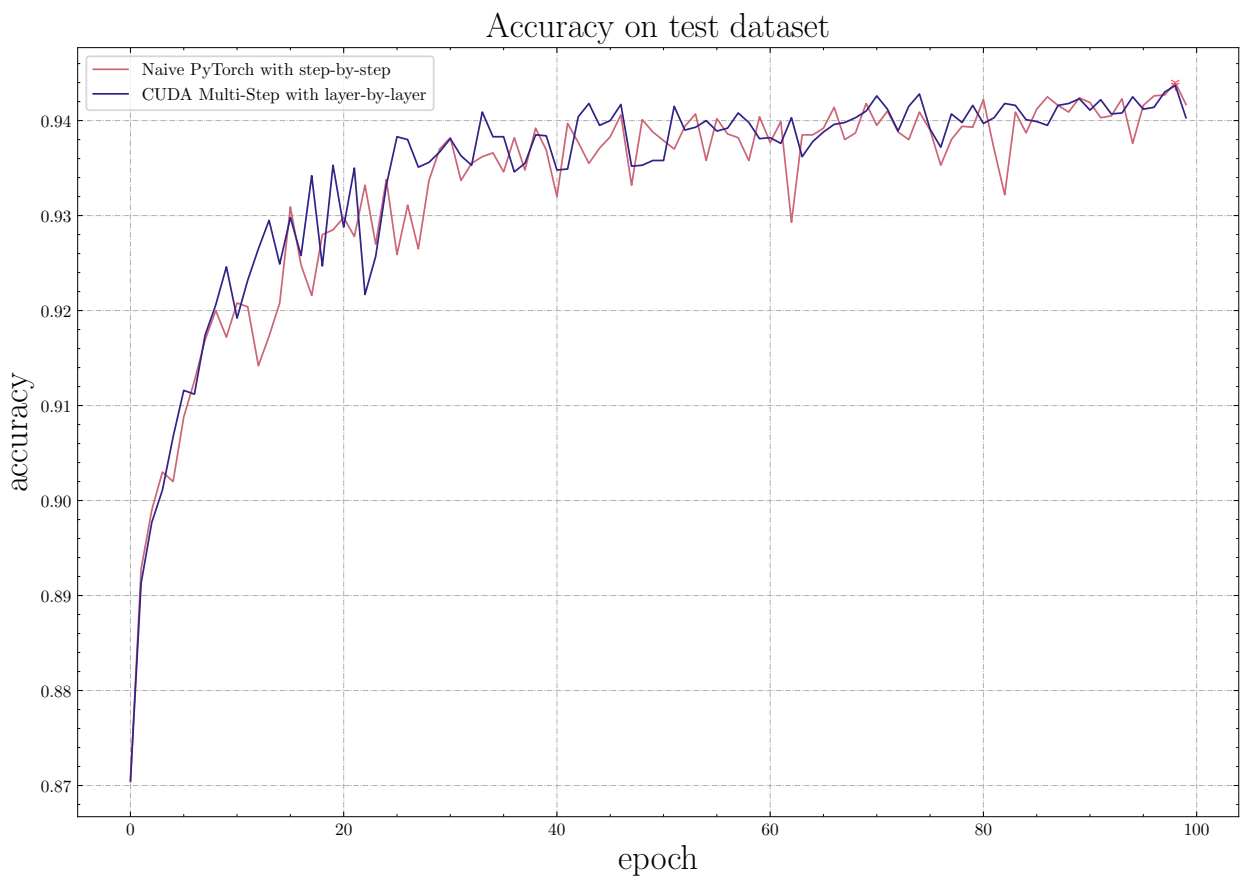
(下页继续)

(续上页)

```
saving net...
saved
epoch=2, t_train=26.281963238492608, t_test=1.4901704853400588, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2,␣
↪max_test_accuracy=0.8977, train_times=1404
saving net...
saved

...

epoch=96, t_train=26.286096683703363, t_test=1.5033660298213363, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2,␣
↪max_test_accuracy=0.9428, train_times=45396
saving net...
saved
epoch=97, t_train=26.185854725539684, t_test=1.4934641849249601, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2,␣
↪max_test_accuracy=0.943, train_times=45864
saving net...
saved
epoch=98, t_train=26.256993867456913, t_test=1.5093903196975589, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2,␣
↪max_test_accuracy=0.9437, train_times=46332
epoch=99, t_train=26.200945735909045, t_test=1.4959839908406138, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2,␣
↪max_test_accuracy=0.9437, train_times=46800
```

最终的正确率是 94.37%，与*使用 CUDA 增强的神经元与逐层传播进行加速* 中的 94.4% 相差无几，两者在训练过程中的训练 batch 正确率和测试集正确率曲线如下：


Accuracy on train batch
accuracy
iteration
1.0
0.8
0.6
0.4
0.2
0
10000
20000
30000
40000
Naive PyTorch with step-by-step
CUDA Multi-Step with layer-by-layer



两个网络使用了完全相同的随机种子，最终的性能略有差异，可能是 CUDA 和 PyTorch 的计算数值误差导致的。在日志中记录了训练和测试所需要的时间，我们可以发现，训练耗时为原始网络的 64%，推理耗时为原始网络的 58%，速度有了明显提升。

1.14 监视器

本教程作者：Yanqi-Chen

本节教程将介绍如何使用监视器监视网络状态与高阶统计数据。

1.14.1 监视网络脉冲

首先我们搭建一个简单的两层网络

```
import torch
from torch import nn
from spikingjelly.clock_driven import neuron

net = nn.Sequential(
```

(下页继续)

(续上页)

```
                nn.Linear(784, 100, bias=False),
                neuron.IFNode(),
                nn.Linear(100, 10, bias=False),
                neuron.IFNode()
            )
```

在网络运行之前，我们先创建一个监视器。注意监视器除去网络之外，还有 `device` 和 `backend` 两个参数，可以指定用 numpy 数组或者 PyTorch 张量记录结果并计算。此处我们用 PyTorch 后端，CPU 进行处理。

```
from spikingjelly.clock_driven.monitor import Monitor
mon = Monitor(net, device='cpu', backend='torch')
```

这样就将一个网络与监视器绑定了起来。但是此时监视功能还处于默认的禁用模式，因此在开始记录之前需要手动启用监视功能：

```
mon.enable()
```

给网络以随机的输入 $X \sim \mathcal{N}(1, 4)$

```
neuron_num = 784
T = 8
batch_size = 3
x = torch.rand([T, batch_size, neuron_num]) * 2 + 1
x = x.cuda()

for t in range(T):
    net(x[t])
```

运行结束之后，可以通过监视器获得网络各层神经元的输出脉冲原始数据。Monitor 的 `s` 成员记录了脉冲，为一个以网络层名称为键的字典，每个键对应的的值为一个长为 `T` 的列表，列表中的元素是尺寸为 `[batch_size, ...(神经元尺寸)]` 形状的张量。

在使用结束之后，如果需要清空数据进行下一轮记录，需要使用 `reset` 方法清空已经记录的数据。

```
mon.reset()
```

如果不再需要监视器记录数据（如仅在测试时记录，训练时不记录），可调用 `disable` 方法暂停记录。

```
mon.disable()
```

暂停后监视器仍然与网络绑定，可在下次需要记录时通过 `enable` 方法重新启用。

1.14.2 监视多步网络

监视器同样支持多步模块，在使用上没有区别

```
import torch
from torch import nn
from spikingjelly.cext import neuron as cext_neuron
from spikingjelly.clock_driven import layer

neuron_num = 784
T = 8
batch_size = 3
x = torch.rand([T, batch_size, neuron_num]) * 2 + 1
x = x.cuda()

net = nn.Sequential(
                layer.SeqToANNContainer(
                    nn.Linear(784, 100, bias=False)
                ),
                cext_neuron.MultiStepIFNode(alpha=2.0),
                layer.SeqToANNContainer(
                    nn.Linear(100, 10, bias=False)
                ),
                cext_neuron.MultiStepIFNode(alpha=2.0),
            )

mon = Monitor(net, 'cpu', 'torch')
mon.enable()
net(x)
```

1.14.3 高阶统计数据

目前，监视器支持计算神经元层的**平均发放率**与**未发放神经元占比**两个指标。使用者既可以指定某一层的名称（按照 PyTorch 的模块名称字符串）也可以指定所有层的数据。以对前述的单步网络计算平均发放率为例：

指定参数 `all=True` 为时，记录所有层的平均发放率：

```
print(mon.get_avg_firing_rate(all=True)) # tensor(0.2924)
```

也可以具体到记录某一层：

```
print(mon.get_avg_firing_rate(all=False, module_name='1')) # tensor(0.3183)
print(mon.get_avg_firing_rate(all=False, module_name='3')) # tensor(0.0333)
```

1.15 神经形态数据集处理

本教程作者：fangwei123456

`spikingjelly.datasets` 中集成了常用的神经形态数据集，包括 N-MNIST[1], CIFAR10-DVS[2], DVS128 Gesture[3], NAV Gesture[4], ASLDVS[5] 等。在本节教程中，我们将以 DVS128 Gesture 为例，展示如何使用惊蜇框架处理神经形态数据集。

1.15.1 下载 DVS128 Gesture

DVS128 Gesture 数据集可以从 https://ibm.ent.box.com/s/3hiq58ww1pbbjrinh367ykfdf60xsfm8/folder/50167556794 进行下载。box 网站不支持在不登陆的情况下使用代码直接下载，因此用户需要手动从网站上下载。将数据集下载到了 `E:/datasets/DVS128Gesture`，下载完成后这个文件夹的目录结构为

```
.
|-- DvsGesture.tar.gz
|-- LICENSE.txt
|-- README.txt
`-- gesture_mapping.csv
```

1.15.2 获取 Event 数据

导入惊蜇框架的 DVS128 Gesture 的模块，创建训练集和测试集，其中参数 `use_frame=False` 表示我们不使用帧数据，而是使用 Event 数据。

```
from spikingjelly.datasets import DVS128Gesture

root_dir = 'E:/datasets/DVS128Gesture'
train_set = DVS128Gesture(root_dir, train=True, use_frame=False)
test_set = DVS128Gesture(root_dir, train=True, use_frame=False)
```

运行这段代码，惊蜇框架将会完成以下工作：

1. 检测数据集是否存在，如果存在，则进行 MD5 校验，确认数据集无误后，开始进行解压。将原始数据解压到同级目录下的 `extracted` 文件夹

[1] Orchard, Garrick, et al. "Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades." Frontiers in Neuroscience, vol. 9, 2015, pp. 437–437.

[2] Li, Hongmin, et al. "CIFAR10-DVS: An Event-Stream Dataset for Object Classification." Frontiers in Neuroscience, vol. 11, 2017, pp. 309–309.

[3] Amir, Arnon, et al. "A Low Power, Fully Event-Based Gesture Recognition System." 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 7388–7397.

[4] Maro, Jean-Matthieu, et al. "Event-Based Visual Gesture Recognition with Background Suppression Running on a Smart-Phone." 2019 14th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2019), 2019, p. 1.

[5] Bi, Yin, et al. "Graph-Based Object Classification for Neuromorphic Vision Sensing." 2019 IEEE/CVF International Conference on Computer Vision (ICCV), 2019, pp. 491–501.

#. DVS128 Gesture 中的每个样本，是在不同光照环境下，对不同表演者进行录制的手势视频。一个 AER 文件中包含了多个手势，对应的会有一个 csv 文件来标注整个视频内各个时间段内都是哪种手势。因此，单个的视频文件并不是一个类别，而是多个类别的集合。惊蜇框架会启动多线程进行划分，将每个视频中的每个手势类别文件单独提取出来

下面是运行过程中的命令行输出：

```
DvsGesture.tar.gz already exists, check md5
C:/Users/fw/anaconda3/envs/pytorch-env/lib/site-packages/torchaudio/backend/utils.
↪py:88: UserWarning: No audio backend is available.
  warnings.warn('No audio backend is available.')
md5 checked, extracting...
mkdir E:/datasets/DVS128Gesture/events_npy
mkdir E:/datasets/DVS128Gesture/events_npy/train
mkdir E:/datasets/DVS128Gesture/events_npy/test
read events data from *.aedat and save to *.npy...
convert events data from aedat to numpy format.
thread 0 start
thread 1 start
thread 2 start
thread 3 start
thread 4 start
thread 5 start
thread 6 start
thread 7 start
  0%|          | 0/122 [00:00<?, ?it/s]working thread: [0, 1, 2, 3, 4, 5, 6, 7]
finished thread: []
```

提取各个手势类别的速度较慢，需要耐心等待。运行完成后，同级目录下会多出一个 `events_npy` 文件夹，其中包含训练集和测试集：

```
|-- events_npy
|   |-- test
|   `-- train
```

打印一个数据：

```
x, y = train_set[0]
print('event', x)
print('label', y)
```

得到输出为：

```
event {'t': array([172843814, 172843824, 172843829, ..., 179442748, 179442763,
       179442789]), 'x': array([ 54,  59,  53, ...,  36, 118, 118]), 'y': array([116,␣
↪113,  92, ..., 102,  80,  83]), 'p': array([0, 1, 1, ..., 0, 1, 1])}
```

(下页继续)

(续上页)

```
label 9
```

其中 x 使用字典格式存储 Events 数据，键为 `['t', 'x', 'y', 'p']`；y 是数据的标签，DVS128 Gesture 共有 11 类。

1.15.3 获取 Frame 数据

将原始的 Event 流积分成 Frame 数据，是常用的处理方法，我们采用[6] 的实现方式。。我们将原始的 Event 数据记为 $E(x_i, y_i, t_i, p_i), 0 \leq i \leq N$；设置 `split_by='number'` 表示从 Event 数量 N 上进行划分，接近均匀地划分为 `frames_num=20`，也就是 T 段。记积分后的 Frame 数据中的某一帧为 $F(j)$，在:math‘(p, x, y)‘ 位置的像素值为 $F(j, p, x, y)$；math:*F(j)* 是从 Event 流中索引介于 j_l 和 j_r 的 Event 积分而来：

$$
\begin{aligned}
j_l &= \left\lfloor \frac{N}{T} \right\rfloor \cdot j \\
j_r &= \begin{cases} \left\lfloor \frac{N}{T} \right\rfloor \cdot (j+1), & \text{if } j < T-1 \\ N, & \text{if } j = T-1 \end{cases} \\
F(j, p, x, y) &= \sum_{i=j_l}^{j_r - 1} \mathcal{I}_{p,x,y}(p_i, x_i, y_i)
\end{aligned}
$$

其中 $\lfloor \cdot \rfloor$ 是向下取整，$\mathcal{I}_{p,x,y}(p_i, x_i, y_i)$ 是示性函数，当且仅当 $(p, x, y) = (p_i, x_i, y_i)$ 时取值为 1，否则为 0。

运行下列代码，惊蛰框架就会开始进行积分，创建 Frame 数据集：

```
train_set = DVS128Gesture(root_dir, train=True, use_frame=True, frames_num=20, split_
↪by='number', normalization=None)
test_set = DVS128Gesture(root_dir, train=True, use_frame=True, frames_num=20, split_
↪by='number', normalization=None)
```

命令行的输出为：

```
npy format events data root E:/datasets/DVS128Gesture/events_npy/train, E:/datasets/
↪DVS128Gesture/events_npy/test already exists
mkdir E:/datasets/DVS128Gesture/frames_num_20_split_by_number_normalization_None, E:/
↪datasets/DVS128Gesture/frames_num_20_split_by_number_normalization_None/train, E:/
↪datasets/DVS128Gesture/frames_num_20_split_by_number_normalization_None/test.
creating frames data..
thread 0 start, processing files index: 0 : 294.
thread 1 start, processing files index: 294 : 588.
thread 2 start, processing files index: 588 : 882.
thread 4 start, processing files index: 882 : 1176.
thread 0 finished.
```

(下页继续)

[6] Fang, Wei, et al. “Incorporating Learnable Membrane Time Constant to Enhance Learning of Spiking Neural Networks.” ArXiv: Neural and Evolutionary Computing, 2020.

(续上页)

```
thread 1 finished.
thread 2 finished.
thread 3 finished.
thread 0 start, processing files index: 0 : 72.
thread 1 start, processing files index: 72 : 144.
thread 2 start, processing files index: 144 : 216.
thread 4 start, processing files index: 216 : 288.
thread 0 finished.
thread 1 finished.
thread 2 finished.
thread 3 finished.
```

运行后，同级目录下会出现 `frames_num_20_split_by_number_normalization_None` 文件夹，这里存放了积分生成的 Frame 数据。

打印一个数据：

```
x, y = train_set[0]
x, y = train_set[0]
print('frame shape', x.shape)
print('label', y)
```

得到输出为：

```
frame shape torch.Size([20, 2, 128, 128])
label 9
```

查看 1 个积分好的 Frame 数据：

```
from torchvision import transforms
from matplotlib import pyplot as plt

x, y = train_set[5]
to_img = transforms.ToPILImage()

img_tensor = torch.zeros([x.shape[0], 3, x.shape[2], x.shape[3]])
img_tensor[:, 1] = x[:, 0]
img_tensor[:, 2] = x[:, 1]


for t in range(img_tensor.shape[0]):
    print(t)
    plt.imshow(to_img(img_tensor[t]))
    plt.pause(0.01)
```

显示效果如下图所示：

CHAPTER 2

模块文档

- *APIs*

CHAPTER 3

文档索引

- genindex
- modindex
- search

CHAPTER 4

引用

如果您在自己的工作中用到了惊蜇 (SpikingJelly)，您可以按照下列格式进行引用：

```
@misc{SpikingJelly,
    title = {SpikingJelly},
    author = {Fang, Wei and Chen, Yanqi and Ding, Jianhao and Chen, Ding and Yu,␣
↪Zhaofei and Zhou, Huihui and Tian, Yonghong and other contributors},
    year = {2020},
    publisher = {GitHub},
    journal = {GitHub repository},
    howpublished = {\url{https://github.com/fangwei123456/spikingjelly}},
}
```

CHAPTER 5

项目信息

北京大学信息科学技术学院数字媒体所媒体学习组 Multimedia Learning Group 和 鹏城实验室 是 SpikingJelly 的主要开发者。

开发人员名单可见于 贡献者 。

CHAPTER 6

Welcome to SpikingJelly’s documentation

SpikingJelly is an open-source deep learning framework for Spiking Neural Network (SNN) based on PyTorch.

- 中文首页

6.1 Installation

Note that SpikingJelly is based on PyTorch. Please make sure that you have installed PyTorch before you install SpikingJelly.

Install from PyPI:

```
pip install spikingjelly
```

Note that the CUDA extensions are not included in the PyPI package. If you want to use the CUDA extensions, please **install from the source codes**:

From GitHub:

```
git clone https://github.com/fangwei123456/spikingjelly.git
cd spikingjelly
python setup.py install
```

From OpenI:

```
git clone https://git.openi.org.cn/OpenI/spikingjelly.git
cd spikingjelly
python setup.py install
```

6.1.1 Clock driven: Neurons

Author: fangwei123456

Translator: YeYumin

This tutorial focuses on `spikingjelly.clock_driven.neuron`, which introduces spiking neurons, and clock-driven simulation methods.

Spiking neuron model

In `spikingjelly`, we define a neuron which can only output spikes, i.e. 0 or 1, as a "spiking neuron". Networks that use spiking neurons are called Spiking Neural Networks. `spikingjelly.clock_driven.neuron` defines various common spiking neuron models. We take `spikingjelly.clock_driven.neuron.LIFNode` as an example to introduce spiking neurons.

First, we need to import the relevant modules:

```
import torch
import torch.nn as nn
import numpy as np
from spikingjelly.clock_driven import neuron
from spikingjelly import visualizing
from matplotlib import pyplot as plt
```

And then create a new LIF neuron layer:

```
lif = neuron.LIFNode()
```

The LIF neuron layer has some parameters, which are explained in detail in the API documentation:

- **tau** –membrane potential time constant, which is shared by all neurons in this layer
- **v_threshold** –the threshold voltage of the neuron
- **v_reset** –the reset voltage of the neuron. If it is not `None`, when the neuron releases a spike, the voltage will be reset to `v_reset`; if it is set to `None`, the voltage will be subtracted from `v_threshold`
- **surrogate_function** –the surrogate function used to calculate the gradient of the spike function during back propagation
- **monitor_state** –whether to set up a monitor to save the voltages and spikes of the neurons. If it is `True`,

`self.monitor` is a dictionary, the keys include `h`, `v` and `s`, which record the voltage after charging, the voltage after releasing a spike, and the released spike respectively. The corresponding value is a linked list. In order to save memory, the value stored in the list is the value of the original variable converted into a `numpy` array. Also note that the `self.reset()` function will clear these linked lists.

The `surrogate_function` behaves exactly the same as the step function during forward propagation, and we will introduce its working principle for back propagation later.

You may be curious about the number of neurons in this layer. For most neuron layers in `spikingjelly.clock_driven.neuron`, the number of neurons is automatically determined according to the `shape` of the received input after initialization or re-initialization by calling the `reset()` function.

Very similar to neurons in RNN, spiking neurons are also stateful, i.e., they have memory. The state variable of a spiking neuron is generally its membrane potential V_t. Therefore, neurons in `spikingjelly.clock_driven.neuron` have state variable `v`. You can print out the membrane potential of the newly created LIF neuron layer:

```
print(lif.v)
# 0.0
```

You can find that `v` is now `0.0` because we haven't given it any input yet. We apply several different inputs and observe the `shape` of the voltage of the neuron, which is consistent with the number of neurons:

```
x = torch.rand(size=[2, 3])
lif(x)
print('x.shape', x.shape, 'lif.v.shape', lif.v.shape)
# x.shape torch.Size([2, 3]) lif.v.shape torch.Size([2, 3])
lif.reset()

x = torch.rand(size=[4, 5, 6])
lif(x)
print('x.shape', x.shape, 'lif.v.shape', lif.v.shape)
# x.shape torch.Size([4, 5, 6]) lif.v.shape torch.Size([4, 5, 6])
lif.reset()
```

So what is the relationship between V_t and input X_t? In a spiking neuron, it not only depends on the input X_t at the current moment, but also on its membrane potential V_{t-1} at the end of the previous moment.

Usually we use the sub-threshold (when the membrane potential does not exceed the threshold voltage `V_{threshold}`) charging differential equation $\frac{\mathrm{d}V(t)}{\mathrm{d}t} = f(V(t), X(t))$ to describe the continuous-time spiking neuron charging process. For example, for LIF neurons, the charging equation is:

$$\tau_m \frac{\mathrm{d}V(t)}{\mathrm{d}t} = -(V(t) - V_{reset}) + X(t)$$

Where τ_m is the membrane potential time constant and V_{reset} is the reset voltage. For such differential equations, since $X(t)$ is not a constant, it is difficult to obtain a explicit analytical solution.

The neurons in `spikingjelly.clock_driven.neuron` use discrete difference equations to approximate continuous differential equations. From the perspective of the difference equation, the charging equation of the LIF neuron

is:

$$\tau_m(V_t - V_{t-1}) = -(V_{t-1} - V_{reset}) + X_t$$

Therefore, the expression of V_t can be obtained as

$$V_t = f(V_{t-1}, X_t) = V_{t-1} + \frac{1}{\tau_m}(-(V_{t-1} - V_{reset}) + X_t)$$

The corresponding code can be found in `neuronal_charge()` of `LIFNode`:

```
def neuronal_charge(self, dv: torch.Tensor):
    if self.v_reset is None:
        self.v += (dv - self.v) / self.tau
    else:
        self.v += (dv - (self.v - self.v_reset)) / self.tau
```

Different neurons have different charging equations. However, when the membrane potential exceeds the threshold voltage, the release of a spike and the reset of the membrane potential after releasing a spike are the same. Therefore, they all inherit from `BaseNode` and share the same discharge and reset equations. The code for releasing a spike can be found in `neuronal_fire()` of `BaseNode`:

```
def neuronal_fire(self):
    self.spike = self.surrogate_function(self.v - self.v_threshold)
```

`surrogate_function()` is a step function during forward propagation, as long as the input is greater than or equal to 0, it will return 1, otherwise it will return 0. We regard this kind of `tensor` whose elements are only 0 or 1 as spikes.

The release of a spike consumes the previously accumulated electric charge of the neuron, so there will be an instantaneous decrease in the membrane potential, which is the reset of the membrane potential. In SNN, there are two ways to realize membrane potential reset:

1. Hard method: After releasing a spike, the membrane potential is directly set to the reset voltage:$V = V_{reset}$
2. Soft method: After releasing a spike, the membrane potential subtracts the threshold voltage:$V = V - V_{threshold}$

It can be found that for neurons using the Soft method, there is no need to reset the voltage V_{reset}. For the neuron in `spikingjelly.clock_driven.neuron`, when `v_reset` is set to the default value `1.0`, the neuron uses the Hard mode; if it is set to `None`, the Soft mode will be used. You can find the corresponding code in `neuronal_reset()` of `BaseNode`:

```
def neuronal_reset(self):
    if self.detach_reset:
        spike = self.spike.detach()
    else:
        spike = self.spike

    if self.v_reset is None:
```

(下页继续)

(续上页)

```
        self.v = self.v - spike * self.v_threshold
    else:
        self.v = (1 - spike) * self.v + spike * self.v_reset
```

Three equations describing discrete spiking neurons

So far, we can use the three discrete equations of charging, discharging, and resetting to describe any discrete spiking neurons. The charging and discharging equations are:

$$
\begin{aligned}
H_t &= f(V_{t-1}, X_t) \\
S_t &= g(H_t - V_{threshold}) = \Theta(H_t - V_{threshold})
\end{aligned}
$$

where $\Theta(x)$ is the `surrogate_function()` in the parameter list, which is a step function:

$$
\Theta(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}
$$

The hard reset equation is:

$$
V_t = H_t \cdot (1 - S_t) + V_{reset} \cdot S_t
$$

The soft reset equation is:

$$
V_t = H_t - V_{threshold} \cdot S_t
$$

where V_t is the membrane potential of the neuron, X_t is the external input, such as voltage increment. To avoid confusion, we use H_t to represent the membrane potential before the neuron releases a spike, V_t is the membrane potential after the neuron releases a spike, $f(V(t-1), X(t))$ is the update equation of the neuronal state. The difference between different neurons is the update equation.

Clock-driven simulation

`spikingjelly.clock_driven` uses a clock-driven approach to gradually simulate SNN.

Next, we will gradually stimulate the neuron and check its membrane potential and output spikes. In order to record the data, we need to open the `monitor` of the neuron layer:

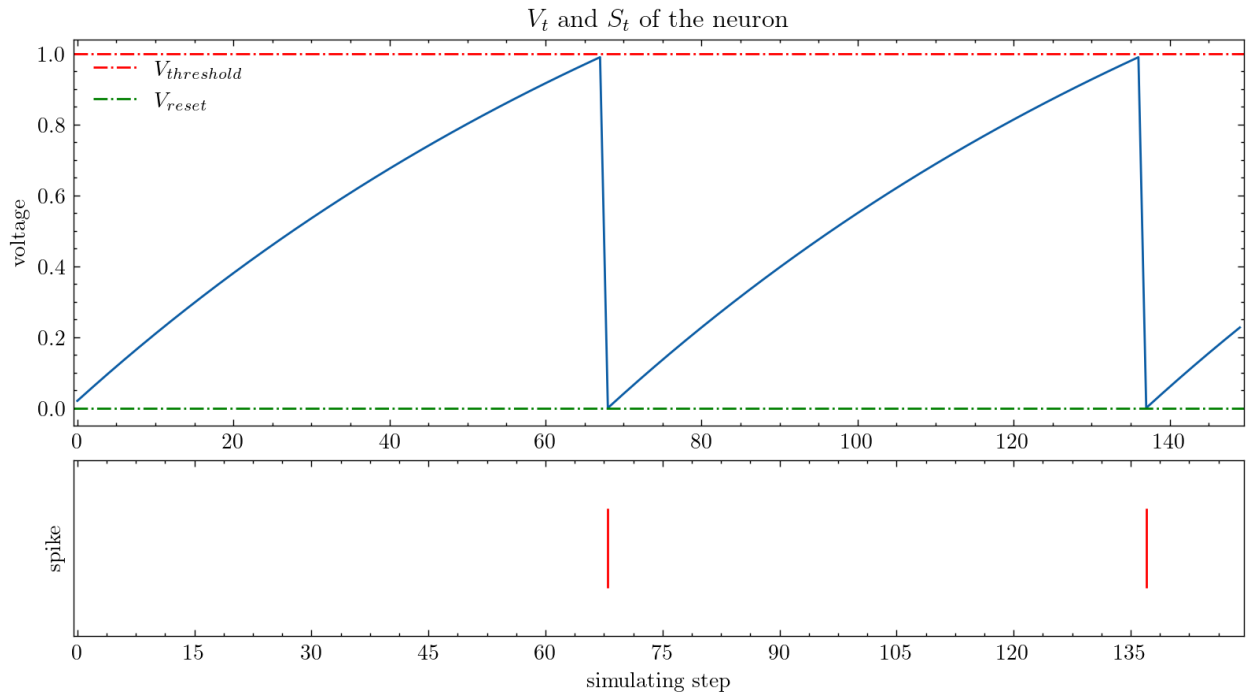
```
lif.set_monitor(True)
```

After turning on the monitor, the neuron layer will automatically record the charged membrane potential `self.monitor['h']`, the output spikes `self.monitor['s']`, and the membrane potential after discharging `self.monitor['v']` in the dictionary `self.monitor` during simulation.

Now let us exert continuous inputs to the LIF neuron layer and plot the membrane potential and output spikes:

```
x = torch.Tensor([2.0])
T = 150
for t in range(T):
    lif(x)
visualizing.plot_one_neuron_v_s(lif.monitor['v'], lif.monitor['s'], v_threshold=lif.v_
↪threshold, v_reset=lif.v_reset, dpi=200)
plt.show()
```

We exert an input with `shape=[1]`, so this LIF neuron layer has only 1 neuron. Its membrane potential and output spikes change with time as follows:



In the following, we reset the neuron layer and exert an input with `shape=[32]` to view the membrane potential and output spikes of these 32 neurons:

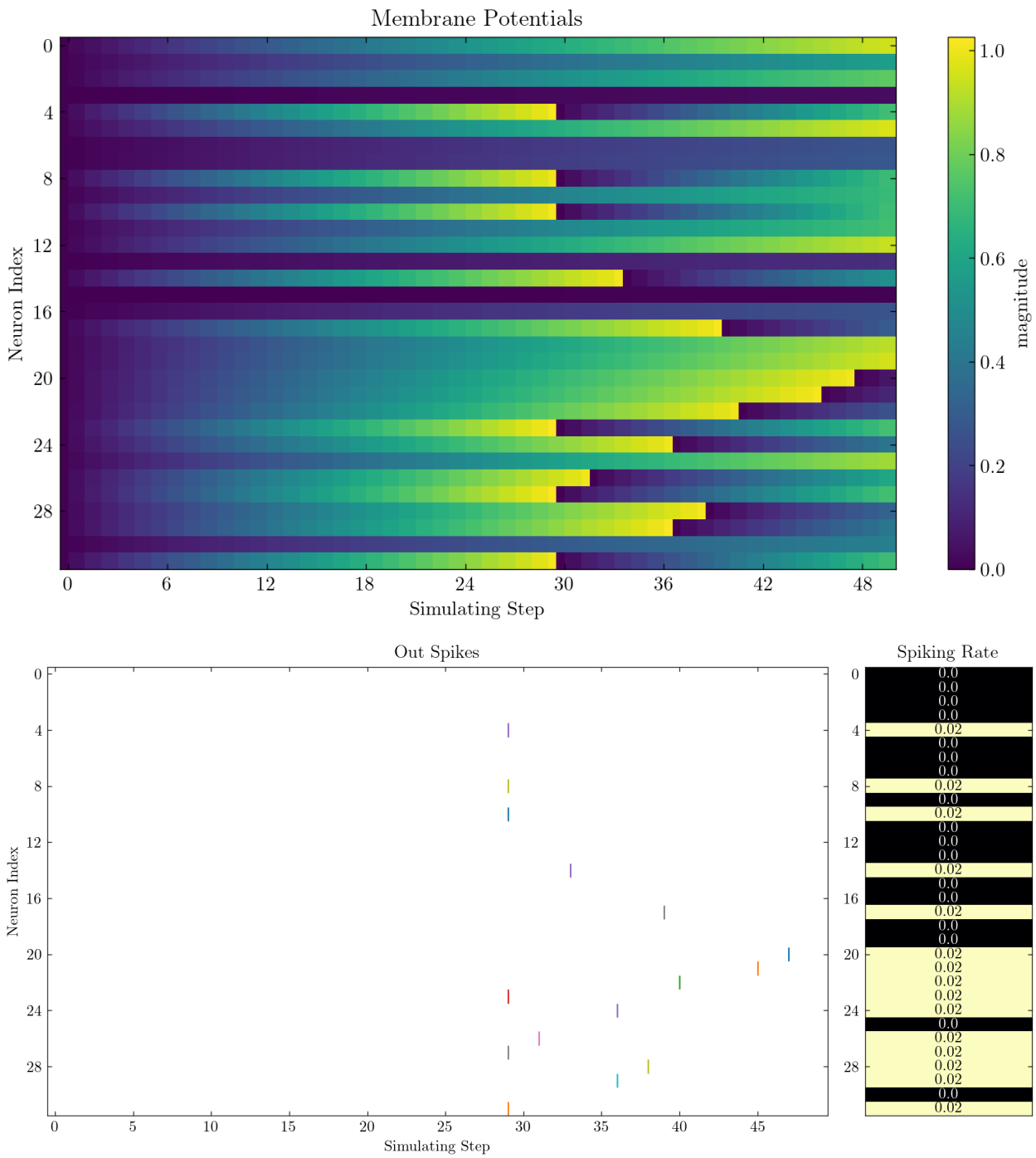
```
lif.reset()
x = torch.rand(size=[32]) * 4
T = 50
for t in range(T):
    lif(x)

visualizing.plot_2d_heatmap(array=np.asarray(lif.monitor['v']).T, title='Membrane␣
↪Potentials', xlabel='Simulating Step',
                                ylabel='Neuron Index', int_x_ticks=True, x_max=T,␣
↪dpi=200)
visualizing.plot_1d_spikes(spikes=np.asarray(lif.monitor['s']).T, title='Membrane␣
↪Potentials', xlabel='Simulating Step',
```

(下页继续)

(续上页)

```
                                    ylabel='Neuron Index', dpi=200)
plt.show()
```

The results are as follows:



6.1.2 Clock driven: Encoder

Author: Grasshlw, Yanqi-Chen

Translator: YeYumin

This tutorial focuses on `spikingjelly.clock_driven.encoding` and introduces several encoders.

The base class of Encoder

In `spikingjelly.clock_driven`, the defined encoders are inherited from the base class of encoder `BaseEncoder`. The `BaseEncoder` inherits `torch.nn.Module`, and defines three methods. The `forward` method converts the input data `x` into spikes. In the `step` method, `x` is encoded into a spike sequence, and `step` is used to obtain the spike data of each step for multiple steps. The `reset` method sets the state variable of an encoder to the initial state.

```
class BaseEncoder(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        raise NotImplementedError

    def step(self):
        raise NotImplementedError

    def reset(self):
        pass
```

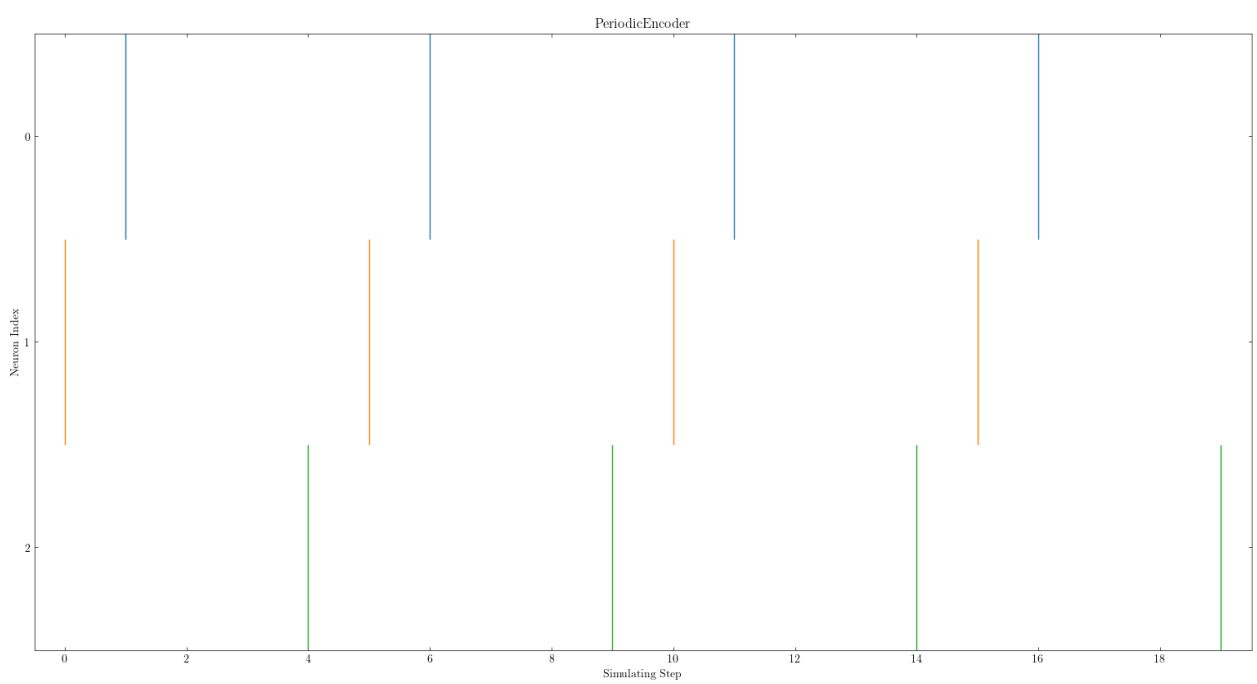
Periodic encoder

Periodic encoder is an encoder that periodically outputs spikes from a given sequence. Regardless of the input data, the class `PeriodicEncoder` has defined the output spike sequence `out_spike` when initialization, and can be reset by the method `set_out_spike` during application.

```
class PeriodicEncoder(BaseEncoder):
    def __init__(self, out_spike):
        super().__init__()
        assert out_spike.dtype == torch.bool
        self.out_spike = out_spike
        self.T = out_spike.shape[0]
        self.index = 0
```

Example: Considering three neurons and spike sequences with 5 time steps, which are `01000`, `10000`, and `00001` respectively, we initialize a periodic encoder and output simulated spike data with 20 time steps.

```
import torch
import matplotlib
import matplotlib.pyplot as plt
from spikingjelly.clock_driven import encoding
from spikingjelly import visualizing

# Given spike sequence
set_spike = torch.full((3, 5), 0, dtype=torch.bool)
set_spike[0, 1] = 1
set_spike[1, 0] = 1
set_spike[2, 4] = 1

pe = encoding.PeriodicEncoder(set_spike.transpose(0, 1))

# Output the coding result of the periodic encoder
out_spike = torch.full((3, 20), 0, dtype=torch.bool)
for t in range(out_spike.shape[1]):
    out_spike[:, t] = pe.step()

plt.style.use(['science', 'muted'])
visualizing.plot_1d_spikes(out_spike.float().numpy(), 'PeriodicEncoder', 'Simulating␣
↪Step', 'Neuron Index',
                           plot_firing_rate=False)
plt.show()
```



Latency encoder

The latency encoder is an encoder that delays the delivery of spikes based on the input data `x`. When the stimulus intensity is greater, the firing time is earlier, and there is a maximum spike latency. Therefore, for each input data `x`, a spike sequence with a period of the maximum spike latency can be obtained.

The spike firing time t_i and the stimulus intensity x_i satisfy the following formulas. When the encoding type is linear (`function_type='linear'`)

$$t_i = (t_{max} - 1) * (1 - x_i)$$

When the encoding type is logarithmic (`function_type='log'`)

$$t_i = (t_{max} - 1) - ln(\alpha * x_i + 1)$$

In the formulas, t_{max} is the maximum spike latency, and x_i needs to be normalized to $[0, 1]$.

Consider the second formula, α needs to satisfy:

$$(t_{max} - 1) - ln(\alpha * 1 + 1) = 0$$

This may cause the encoder to overflow:

$$\alpha = e^{t_{max}-1} - 1$$

because α will increase exponentially as t_{max} increases.

Example: Randomly generate six `x`, each of which is the stimulation intensity of 6 neurons, and set the maximum spike latency to 20, then use `LatencyEncoder` to encode the above input data.

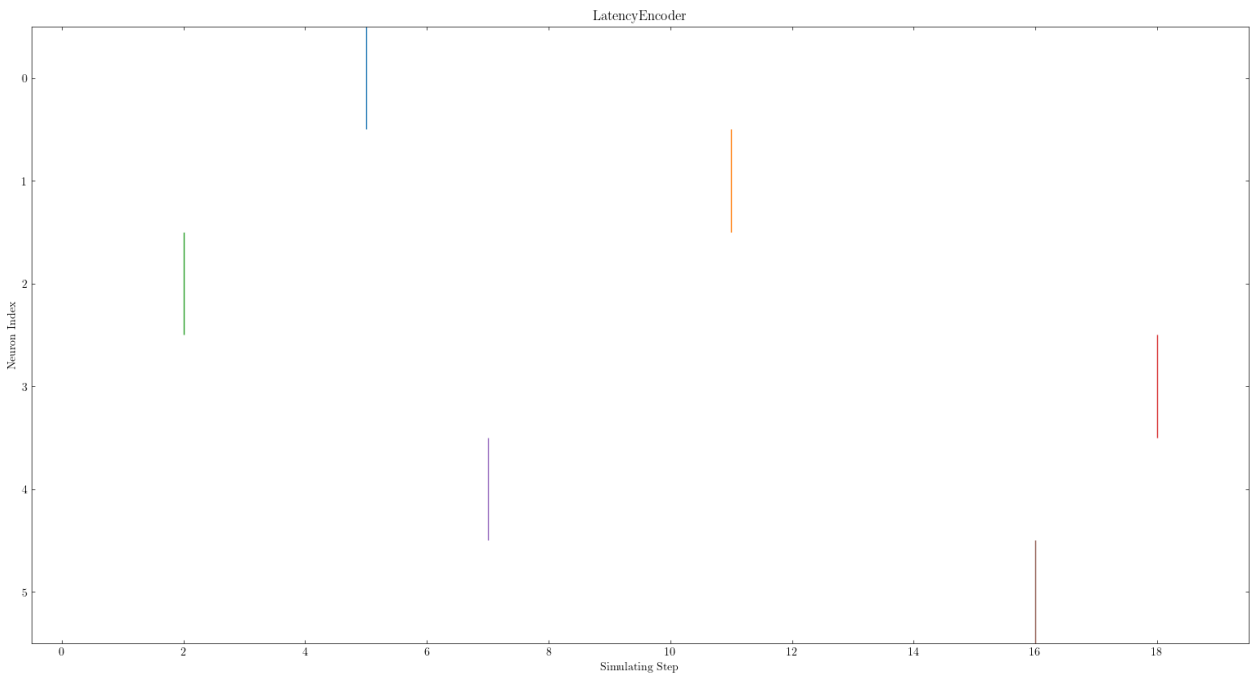
```
import torch
import matplotlib
import matplotlib.pyplot as plt
from spikingjelly.clock_driven import encoding
from spikingjelly import visualizing

# Randomly generate stimulation intensity of 6 neurons, set the maximum spike time to␣
↪20
x = torch.rand(6)
max_spike_time = 20

# Encode input data into spike sequence
le = encoding.LatencyEncoder(max_spike_time)
le(x)

# Output the encoding result of the delayed encoder
out_spike = torch.full((6, 20), 0, dtype=torch.bool)
```

(下页继续)

(续上页)

```
for t in range(max_spike_time):
    out_spike[:, t] = le.step()

print(x)
plt.style.use(['science', 'muted'])
visualizing.plot_1d_spikes(out_spike.float().numpy(), 'LatencyEncoder', 'Simulating␣
↪Step', 'Neuron Index',
                           plot_firing_rate=False)
plt.show()
```

When the randomly generated stimulus intensities are `0.6650`, `0.3704`, `0.8485`, `0.0247`, `0.5589`, and `0.1030`, the spike sequence obtained is as follows:



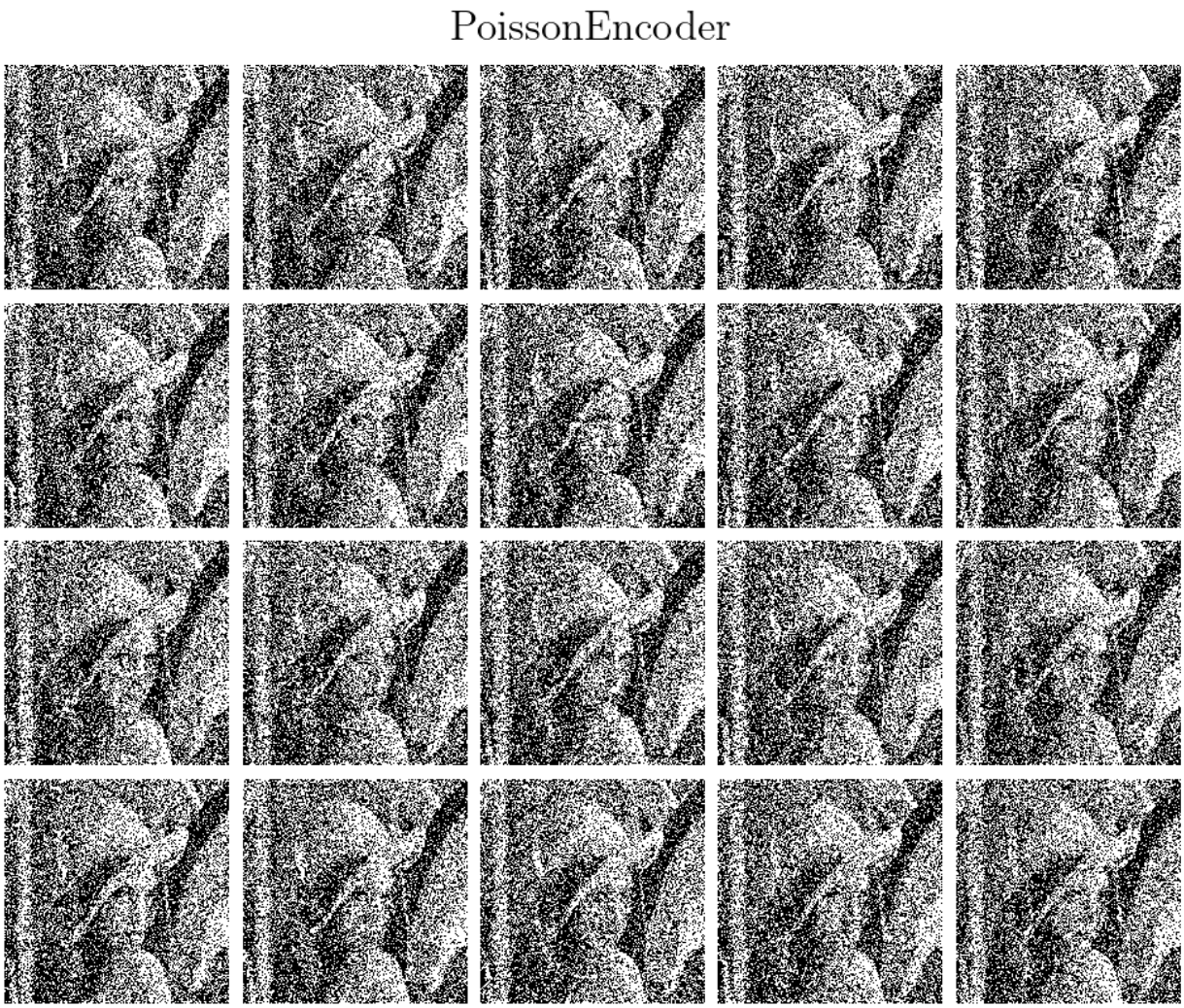
Poisson encoder

The Poisson encoder converts the input data `x` into a spike sequence, which conforms to a Poisson process, i.e., the number of spikes during a certain period follows a Poisson distribution. A Poisson process is also called a Poisson flow. When a spike flow satisfies the requirements of independent increment, incremental stability and commonality, such a spike flow is a Poisson flow. More specifically, in the entire spike stream, the number of spikes appearing in disjoint intervals is independent of each other, and in any interval, the number of spikes is related to the length of the interval while not the starting point of the interval. Therefore, in order to realize Poisson encoding, we set the firing probability of a time step $p = x$, where x needs to be normalized to [0, 1].

Example: The input image is lena512.bmp , and 20 time steps are simulated to obtain 20 spike matrices.

```
import torch
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from PIL import Image
from spikingjelly.clock_driven import encoding
from spikingjelly import visualizing

# Read in Lena image
lena_img = np.array(Image.open('lena512.bmp')) / 255
x = torch.from_numpy(lena_img)

pe = encoding.PoissonEncoder()

# Simulate 20 time steps, encode the image into a spike matrix and output
w, h = x.shape
out_spike = torch.full((20, w, h), 0, dtype=torch.bool)
T = 20
for t in range(T):
    out_spike[t] = pe(x)

plt.figure()
plt.style.use(['science', 'muted'])
plt.imshow(x, cmap='gray')
plt.axis('off')

visualizing.plot_2d_spiking_feature_map(out_spike.float().numpy(), 4, 5, 30,
↪'PoissonEncoder')
plt.axis('off')
```

The original grayscale image of Lena and 20 resulted spike matrices are as follows:



Comparing the original grayscale image to the spike matrix, it can be found that the spike matrix is very close to the contour of the original grayscale image, which shows the superiority of the Poisson encoder.

After simulating the Poisson encoder with the Lena grayscale image for 512 time steps, we superimpose the spike matrix obtained in each step, and obtain the result of the superposition of steps 1, 128, 256, 384, and 512, and draw the picture:

```
# Simulate 512 time steps, superimpose the coded spike matrix one by one to obtain␣
↪the 1, 128, 256, 384, 512th superposition results and output
superposition = torch.full((w, h), 0, dtype=torch.float)
superposition_ = torch.full((5, w, h), 0, dtype=torch.float)
T = 512
for t in range(T):
    superposition += pe(x).float()
    if t == 0 or t == 127 or t == 255 or t == 387 or t == 511:
        superposition_[int((t + 1) / 128)] = superposition
```

(下页继续)

(续上页)

```
# Normalized
for i in range(5):
    min_ = superposition_[i].min()
    max_ = superposition_[i].max()
    superposition_[i] = (superposition_[i] - min_) / (max_ - min_)

# plot
visualizing.plot_2d_spiking_feature_map(superposition_.numpy(), 1, 5, 30,
↪'PoissonEncoder')
plt.axis('off')

plt.show()
```

The superimposed images are as follows:

It can be seen that when the simulation is sufficiently long, the original image can almost be reconstructed with the superimposed images composed of spikes obtained by the Poisson encoder.

Gaussian tuning curve encoder

For input data with `M` features, the Gaussian tuning curve encoder uses `tuning_curve_num` neurons to encode each feature of the input data, and encodes each feature as the firing time of these `tuning_curve_num` neurons. Therefore, the encoder has `M` × `tuning_curve_num` neurons to work properly.

For feature X^i, the value range is $X^i_{min} <= X^i <= X^i_{max}$. According to the maximum and minimum features, the mean and standard deviation of Gaussian curve G^j_i can be calculated as follows:

$$\mu^i_j = x^i_{min} + \frac{2j-3}{2}\frac{x^i_{max} - x^i_{min}}{m-2}, \sigma^i_j = \frac{1}{\beta}\frac{x^i_{max} - x^i_{min}}{m-2}$$

where β is usually 1.5. For one feature, all `tuning_curve_num` Gaussian curves have the same shape, while the axes of symmetry are different.

After the Gaussian curve is generated, the output of the Gaussian function corresponding to each input is calculated, and these outputs are linearly converted into firing timestamps between `[0, max_spike_time - 1]`. In addition, the spikes fired at the last moment are ignored as they never happen.

According to the above steps, the encoding of the input data is completed.

Interval encoder

The interval encoder is an encoder that emits a spike every `T` time steps. The encoder is relatively simple and will not be detailed here.

Weighted phase encoder

Weighted phase encoder is based on binary representations of floats.

Inputs are decomposed to fractional bits and the spikes correspond to the binary value from the leftmost bit to the rightmost bit. Compared to rate coding, each spike in phase coding carries more information. When phase is K, number lies in the interval $[0, 1 - 2^{-K}]$ can be encoded. Example when $K = 8$ in original paper[1] is illustrated here:

Phase (K=8)	1	2	3	4	5	6	7	8
Spike weight $\omega(t)$	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
192/256	1	1	0	0	0	0	0	0
1/256	0	0	0	0	0	0	0	1
128/256	1	0	0	0	0	0	0	0
255/256	1	1	1	1	1	1	1	1

6.1.3 Clock driven: Use single-layer fully connected SNN to identify MNIST

Author: Yanqi-Chen

Translator: YeYumin

This tutorial will introduce how to train a simplest MNIST classification network using encoders and alternative gradient methods.

Build a simple SNN network from scratch

When building a neural network in PyTorch, we can simply use `nn.Sequential` to stack multiple network layers to get a feedforward network. The input data will flow through each network layer in order to get the output.

The MNIST Dateset contains several 8-bit grayscale images with the size of 28×28, which include total of 10 categories from 0 to 9. Taking the classification of MNIST as an example, a simple single-layer ANN network is as follows:

[1] Kim J, Kim H, Huh S, et al. Deep neural networks with weighted spikes[J]. Neurocomputing, 2018, 311: 373-386.

```
net = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 10, bias=False),
    nn.Softmax()
    )
```

We can also use SNN with a completely similar structure for classification tasks. As far as this network is concerned, we only need to remove all the activation functions first, and then add the neurons to the original activation function position. Here we choose the LIF neuron:

```
net = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 10, bias=False),
    neuron.LIFNode(tau=tau)
    )
```

Among them, the membrane potential decay constant τ needs to be set by the parameter `tau`.

Train SNN network

First specify the training parameters and several other configurations

```
device = input('Enter the operating device,e.g.:"cpu" or "cuda:0"\n input device, e.g.
↪, "cpu" or "cuda:0": ')
dataset_dir = input('enter the location of the MNIST data set,e.g.:"./"\n input root␣
↪directory for saving MNIST dataset, e.g., "./": ')
batch_size = int(input('input batch_size, e.g.:"64"\n input batch_size, e.g., "64":
↪'))
learning_rate = float(input('input learning rate,e.g.:"1e-3"\n input learning rate, e.
↪g., "1e-3": '))
T = int(input('enter simulation duration, e.g.:"100"\n input simulating steps, e.g.,
↪"100": '))
tau = float(input('input the time constant of the LIF neuron tau, e.g.:"100.0"\n input␣
↪membrane time constant, tau, for LIF neurons, e.g., "100.0": '))
train_epoch = int(input('enter the number of training rounds, that is, the number of␣
↪times to traverse the training set, e.g.:"100"\n input training epochs, e.g., "100
↪": '))
log_dir = input('enter the location to save the tensorboard log file, e.g.:"./"\n␣
↪input root directory for saving tensorboard logs, e.g., "./": ')
```

The optimizer uses Adam and Poisson encoder to perform spike encoding every time when a picture is input.

```
# Use Adam optimizer
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
```

(下页继续)

(续上页)

```
# Use Poisson encoder
encoder = encoding.PoissonEncoder()
```

The writing of training code needs to follow the following three points:

1. The output of the spiking neuron is binary, and directly using the result of a single run for classification is very susceptible to interference. Therefore, it is generally considered that the output of the spike network is the **firing** frequency (or firing rate) of the output layer over a period of time, and the firing rate indicates the response strength of the category. Therefore, the network needs to run for a period of time, that is, the **average distribution rate** after `T` time is used as the classification basis.

2. The desired result we hope is that except for the correct neuron firing the **highest frequency**, the other neurons **remain silent**. Cross-entropy loss or MSE loss is often used, and here we use MSE loss which have a better actual effect.

3. After each network simulation is over, the network status needs to be **reset**.

Combining the above three points, the code of training loop is as follows:

```
for img, label in train_data_loader:
    img = img.to(device)
    label = label.to(device)
    label_one_hot = F.one_hot(label, 10).float()

    optimizer.zero_grad()

    # Run time of T, out_spikes_counter is the tensor of shape=[batch_size, 10]
    # Record the number of spike firings of 10 neurons in the output layer during the␣
↪entire simulation duration
    for t in range(T):
        if t == 0:
            out_spikes_counter = net(encoder(img).float())
        else:
            out_spikes_counter += net(encoder(img).float())

    # out_spikes_counter / T Obtain the spike firing frequency of 10 neurons in the␣
↪output layer during the simulation time
    out_spikes_counter_frequency = out_spikes_counter / T

    # The loss function is the spike firing frequency of the neurons in the output␣
↪layer, and the MSE of the true category
    # Such a loss function will make the spike firing frequency of the i-th neuron in␣
↪the output layer approach 1 when the category i is input, and the spike firing␣
↪frequency of other neurons will approach 0
    loss = F.mse_loss(out_spikes_counter_frequency, label_one_hot)
    loss.backward()
```

(下页继续)

(续上页)

```
    optimizer.step()
    # After optimizing the parameters once, the state of the network needs to be␣
↪reset, because the neurons of SNN have "memory"
    functional.reset_net(net)
```

The complete code is located in `clock_driven.examples.lif_fc_mnist.py`. In the code, we also use Tensorboard to save training logs. You can run it directly on the Python command line:

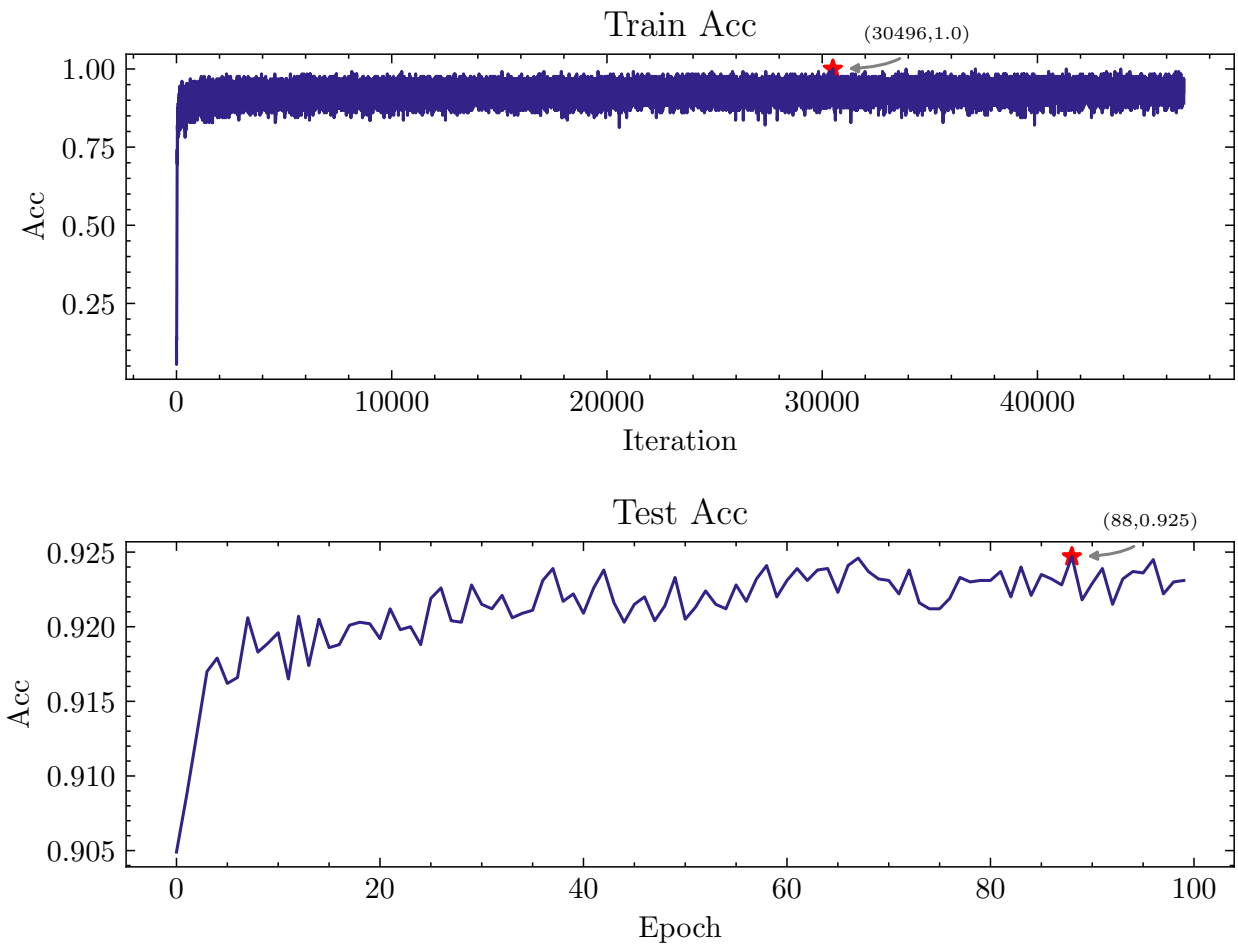
```
>>> import spikingjelly.clock_driven.examples.lif_fc_mnist as lif_fc_mnist
>>> lif_fc_mnist.main()
```

It should be noted that for training such an SNN, the amount of video memory required is linearly related to the simulation duration `T`. A longer `T` is equivalent to using a smaller simulation step, and the training is more "fine", but the training effect is not necessarily better. When `T` is too large, the SNN will become a very deep network after unfolding in time, which will cause the gradient to be easily attenuated or exploded.

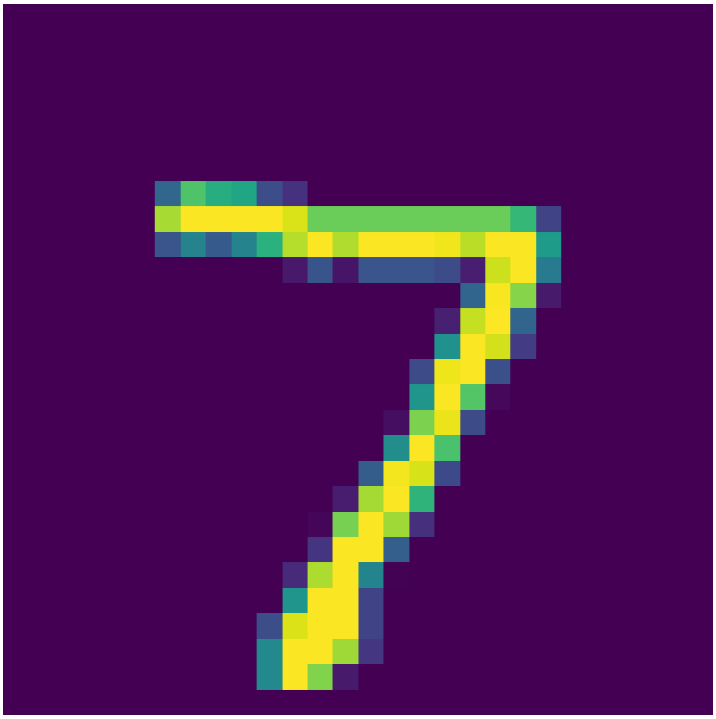
In addition, because we use a Poisson encoder, a larger `T` is required.

Training result

Take `tau=2.0,T=100,batch_size=128,lr=1e-3`, after training 100 Epoch, four npy files will be output. The highest correct rate on the test set is 92.5%, and the correct rate curve obtained through matplotlib visualization is as follows



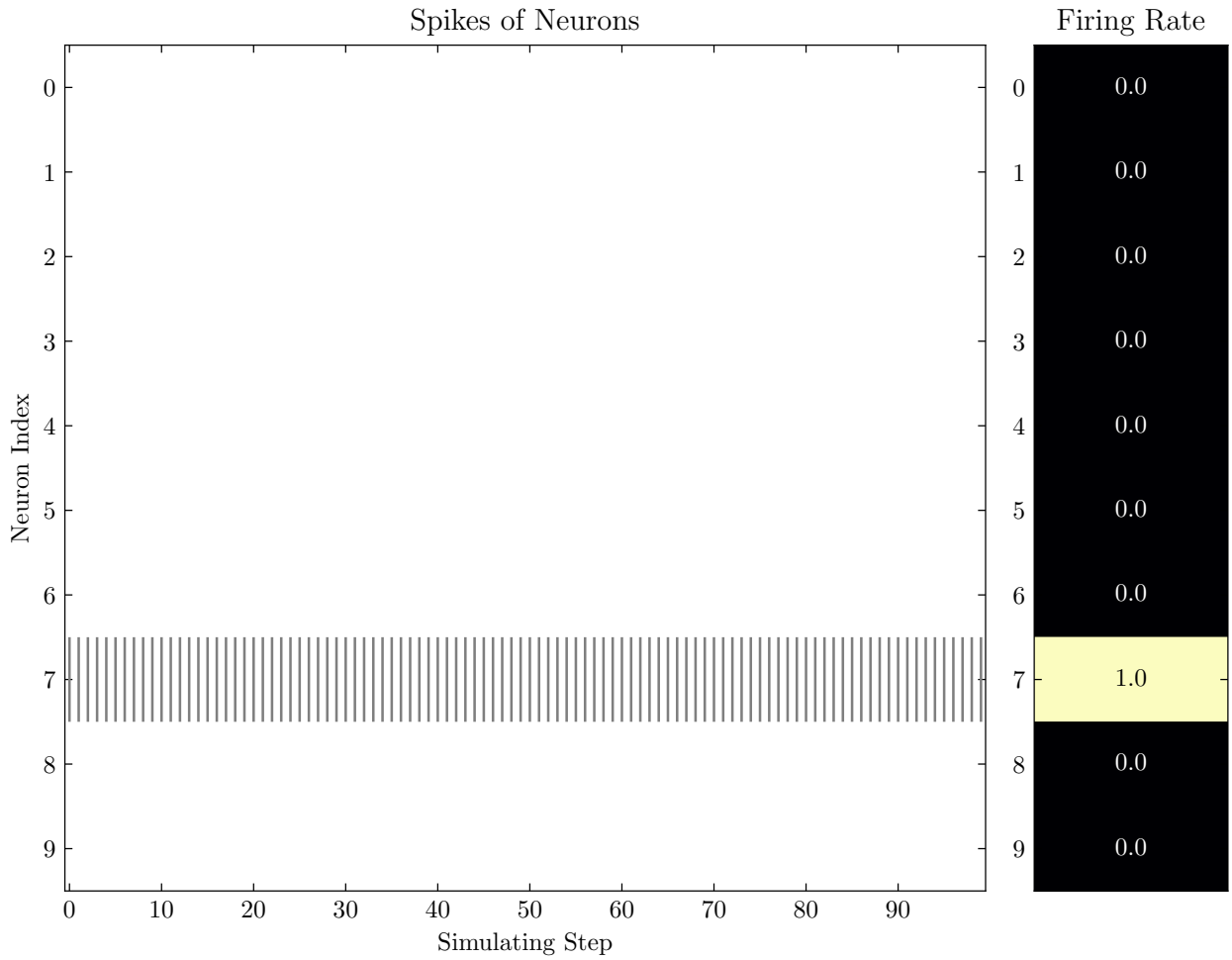
Select the first picture in the test set:

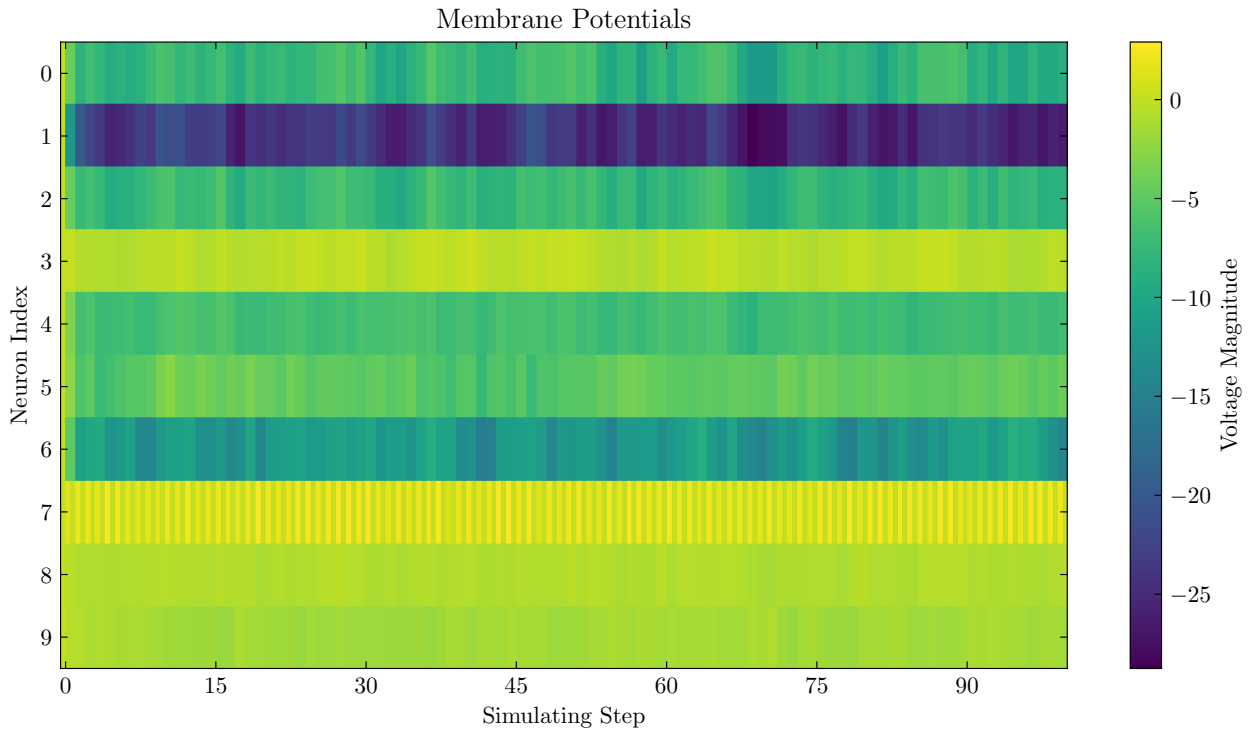


Use the trained model to classify and get the classification result.

```
Firing rate: [[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]]
```

The voltage and spike of the output layer can be visualized by the function in the `visualizing` module as shown in the figure below.


Spikes of Neurons
Firing Rate
Neuron Index
Simulating Step
0
1
2
3
4
5
6
7
8
9
0
10
20
30
40
50
60
70
80
90
0.0
0.0
0.0
0.0
0.0
0.0
0.0
1.0
0.0
0.0



It can be seen that none of the neurons emit any spikes except for the neurons corresponding to the correct category. The complete training code can be found in clock_driven/examples/lif_fc_mnist.py.

6.1.4 Clock driven: Use convolutional SNN to identify Fashion-MNIST

Author: fangwei123456

Translator: YeYumin

In this tutorial, we will build a convolutional spike neural network to classify the Fashion-MNIST dataset. The Fashion-MNIST dataset has the same format as the MNIST dataset, and both are `1 * 28 * 28` grayscale images.

Network structure

Most of the common convolutional neural networks in ANN are in the form of convolution + fully connected layers. We also use a similar structure in SNN. Import related modules, inherit `torch.nn.Module`, and define our network:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
from spikingjelly.clock_driven import neuron, functional, surrogate, layer
from torch.utils.tensorboard import SummaryWriter
import readline
```

(下页继续)

(续上页)

```
class Net(nn.Module):
    def __init__(self, tau, v_threshold=1.0, v_reset=0.0):
```

Then we add a convolutional layer and a fully connected layer to the member variables of `Net`. The developers of `SpikingJelly` found in the experiments that neurons in the convolutional layer is better to use `IFNode` for static image data without time information. We add 2 convolution-BN-pooling layers:

```
self.conv = nn.Sequential(
        nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(128),
        neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
        nn.MaxPool2d(2, 2),  # 14 * 14

        nn.Conv2d(128, 128, kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(128),
        neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
        nn.MaxPool2d(2, 2)  # 7 * 7
    )
```

After the input of `1 * 28 * 28` undergoes such a convolutional layer, an output spike of `128 * 7 * 7` is obtained.

Such a convolutional layer can actually function as an encoder: in the previous tutorial, in the code of MNIST classification, we used a Poisson encoder to encode pictures into spikes. In fact, we can directly send the picture to the SNN. In this case, the first spike neuron layer and the previous layer in the SNN can be regarded as an auto-encoder with learnable parameters. For example, these layers in the convolutional layer we just defined:

```
nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
nn.BatchNorm2d(128),
neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_function=surrogate.
↪ATan())
```

This 3-layer network, which receives pictures as input and outputs spikes, can be regarded as an encoder.

Next, we define a 3-layer fully connected network and output the classification results. The fully connected layer generally functions as a classifier, and the performance of using `LIFNode` will be better. Fashion-MNIST has 10 categories, so the output layer is 10 neurons, in order to reduce over-fitting, we also use `layer.Dropout`. For more information about it, please refer to the API documentation.

```
self.fc = nn.Sequential(
    nn.Flatten(),
    layer.Dropout(0.7),
    nn.Linear(128 * 7 * 7, 128 * 3 * 3, bias=False),
```

(下页继续)

(续上页)

```
    neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
    layer.Dropout(0.7),
    nn.Linear(128 * 3 * 3, 128, bias=False),
    neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
    nn.Linear(128, 10, bias=False),
    neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
)
```

Next, define forward propagation. Forward propagation is very simple, first go through convolution and then go through full connection:

```
def forward(self, x):
    return self.fc(self.conv(x))
```

Avoid repeat computing

We can train this network directly, just like the previous MNIST classification:

```
for img, label in train_data_loader:
    img = img.to(device)
    label = label.to(device)
    label_one_hot = F.one_hot(label, 10).float()

    optimizer.zero_grad()

    # run the time of T, out_spikes_counter is the tensor of shape=[batch_size, 10]
    # record the number of spike firings of 10 neurons in the output layer during the␣
↪entire simulation duration
    for t in range(T):
        if t == 0:
            out_spikes_counter = net(encoder(img).float())
        else:
            out_spikes_counter += net(encoder(img).float())

    # out_spikes_counter / T obtain the spike firing frequency of 10 neurons in the␣
↪output layer during the simulation time
    out_spikes_counter_frequency = out_spikes_counter / T

    # the loss function is the spike firing frequency of the neurons in the output␣
↪layer, and the MSE of the true category
```

(下页继续)

(续上页)

```
    # such a loss function will make the spike firing frequency of the i-th neuron in␣
↪the output layer approach 1 when the category i is input, and the spike firing␣
↪frequency of other neurons will approach 0
    loss = F.mse_loss(out_spikes_counter_frequency, label_one_hot)
    loss.backward()
    optimizer.step()
    # after optimizing the parameters once, the state of the network needs to be␣
↪reset, because the neurons of SNN have "memory"
    functional.reset_net(net)
```

But if we re-examine the structure of the network, we can find that some calculations are repeated, for the first 2 layers of the network, the highlighted part of the following code:

```
self.conv = nn.Sequential(
        nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(128),
        neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
        nn.MaxPool2d(2, 2),  # 14 * 14

        nn.Conv2d(128, 128, kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(128),
        neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
        nn.MaxPool2d(2, 2)  # 7 * 7
    )
```

The input images received by these two layers does not change with `t` , but in the `for` loop, each time `img` will recalculate these two layers to get the same output. We extract these layers and encapsulate the time loop into the network itself to facilitate calculation. The new network structure is fully defined as:

```
class Net(nn.Module):
    def __init__(self, tau, T, v_threshold=1.0, v_reset=0.0):
        super().__init__()
        self.T = T

        self.static_conv = nn.Sequential(
            nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(128),
        )

        self.conv = nn.Sequential(
            neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
```

(下页继续)

(续上页)

```
            nn.MaxPool2d(2, 2),  # 14 * 14

            nn.Conv2d(128, 128, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(128),
            neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
            nn.MaxPool2d(2, 2)  # 7 * 7

        )
        self.fc = nn.Sequential(
            nn.Flatten(),
            layer.Dropout(0.7),
            nn.Linear(128 * 7 * 7, 128 * 3 * 3, bias=False),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset,␣
↪surrogate_function=surrogate.ATan()),
            layer.Dropout(0.7),
            nn.Linear(128 * 3 * 3, 128, bias=False),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset,␣
↪surrogate_function=surrogate.ATan()),
            nn.Linear(128, 10, bias=False),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset,␣
↪surrogate_function=surrogate.ATan()),
        )


    def forward(self, x):
        x = self.static_conv(x)

        out_spikes_counter = self.fc(self.conv(x))
        for t in range(1, self.T):
            out_spikes_counter += self.fc(self.conv(x))

        return out_spikes_counter / self.T
```

For SNN whose input does not change with time, although the SNN is stateful as a whole, the first few layers of the network may not be stateful. We can extract these layers separately and put them out of the time loop to avoid additional calculations .

Training network

The complete code is located in `spikingjelly.clock_driven.examples.conv_fashion_mnist`. It can also be run directly from the command line.The network with the highest accuracy of the test set during the training process will be saved in the same level directory of the `tensorboard` log file. The server for training this network uses *Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz* CPU and *GeForce RTX 2080 Ti* GPU.

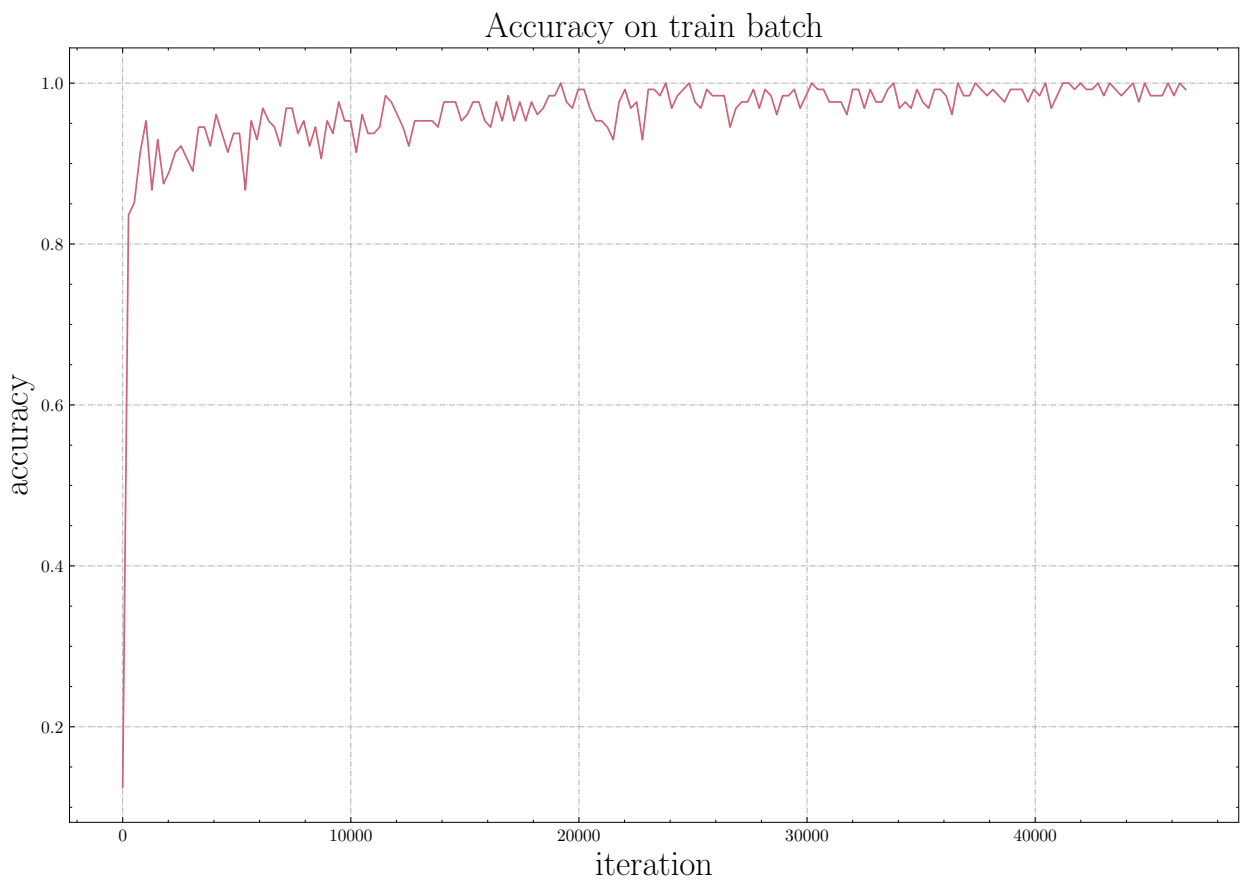
```
>>> from spikingjelly.clock_driven.examples import conv_fashion_mnist
>>> conv_fashion_mnist.main()
输入运行的设备，例如“cpu”或“cuda:0”
 input device, e.g., "cpu" or "cuda:0": cuda:9
输入保存 Fashion MNIST 数据集的位置，例如“./”
 input root directory for saving Fashion MNIST dataset, e.g., "./": ./fmnist
输入 batch_size，例如“64”
 input batch_size, e.g., "64": 128
输入学习率，例如“1e-3”
 input learning rate, e.g., "1e-3": 1e-3
输入仿真时长，例如“8”
 input simulating steps, e.g., "8": 8
输入 LIF 神经元的时间常数 tau，例如“2.0”
 input membrane time constant, tau, for LIF neurons, e.g., "2.0": 2.0
输入训练轮数，即遍历训练集的次数，例如“100”
 input training epochs, e.g., "100": 100
输入保存 tensorboard 日志文件的位置，例如“./”
 input root directory for saving tensorboard logs, e.g., "./": ./logs_conv_fashion_
↪mnist
saving net...
saved
epoch=0, t_train=41.182421264238656, t_test=2.5504338955506682, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_
↪fashion_mnist, max_test_accuracy=0.8704, train_times=468
saving net...
saved
epoch=1, t_train=40.93981215544045, t_test=2.538706629537046, device=cuda:0, dataset_
↪dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_fashion_
↪mnist, max_test_accuracy=0.8928, train_times=936
saving net...
saved
epoch=2, t_train=40.86129532009363, t_test=2.5383697943761945, device=cuda:0, dataset_
↪dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_fashion_
↪mnist, max_test_accuracy=0.899, train_times=1404
saving net...
saved
```

(下页继续)

(续上页)

```
...

epoch=95, t_train=40.98498909268528, t_test=2.558146824128926, device=cuda:0, dataset_
↪dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_fashion_
↪mnist, max_test_accuracy=0.9425, train_times=44928
saving net...
saved
epoch=96, t_train=41.19765609316528, t_test=2.6626883540302515, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_
↪fashion_mnist, max_test_accuracy=0.9426, train_times=45396
saving net...
saved
epoch=97, t_train=41.10238983668387, t_test=2.553960849530995, device=cuda:0, dataset_
↪dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_fashion_
↪mnist, max_test_accuracy=0.9427, train_times=45864
saving net...
saved
epoch=98, t_train=40.89284007716924, t_test=2.5465594390407205, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_
↪fashion_mnist, max_test_accuracy=0.944, train_times=46332
epoch=99, t_train=40.843392613343894, t_test=2.557370903901756, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs_conv_
↪fashion_mnist, max_test_accuracy=0.944, train_times=46800
```

After running 100 rounds of training, the correct rates on the training batch and test set are as follows:


Accuracy on train batch
accuracy
iteration
1.0
0.8
0.6
0.4
0.2
0
10000
20000
30000
40000

After training for 100 epochs, the highest test set accuracy rate can reach 94.4%, which is a very good performance for SNN, only slightly lower than the use of Normalization, random horizontal flip, random vertical flip, random translation in the BenchMark of Fashion-MNIST, ResNet18 of random rotation has a 94.9% correct rate.

Visual encoder

As we said in the previous article, if the data is directly fed into the SNN, the first spike neuron layer and the layers before it can be regarded as a learnable encoder. Specifically, it is the highlighted part of our network as shown below:

```
class Net(nn.Module):
    def __init__(self, tau, T, v_threshold=1.0, v_reset=0.0):
        ...
        self.static_conv = nn.Sequential(
            nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(128),
        )

        self.conv = nn.Sequential(
            neuron.IFNode(v_threshold=v_threshold, v_reset=v_reset, surrogate_
↪function=surrogate.ATan()),
        ...
```

Now let's take a look at the coding effect of the trained encoder. Let's create a new python file, import related modules, and redefine a data loader with `batch_size=1`, because we want to view one picture by one:

```
from matplotlib import pyplot as plt
import numpy as np
from spikingjelly.clock_driven.examples.conv_fashion_mnist import Net
from spikingjelly import visualizing
import torch
import torch.nn as nn
import torchvision

test_data_loader = torch.utils.data.DataLoader(
    dataset=torchvision.datasets.FashionMNIST(
        root=dataset_dir,
        train=False,
        transform=torchvision.transforms.ToTensor(),
        download=True),
    batch_size=1,
    shuffle=True,
    drop_last=False)
```

Load the trained network from the location where the network is saved, that is, under the `log_dir` directory. And we extract the encoder. Just run on the CPU:

```
net = torch.load('./logs_conv_fashion_mnist/net_max_acc.pt', 'cpu')
encoder = nn.Sequential(
    net.static_conv,
    net.conv[0]
)
encoder.eval()
```

Next, extract a picture from the data set, send it to the encoder, and check the accumulated value $\sum_t S_t$ of the output spike. In order to display clearly, we also normalized the pixel value of the output `feature_map`, and linearly transformed the value range to `[0, 1]`.

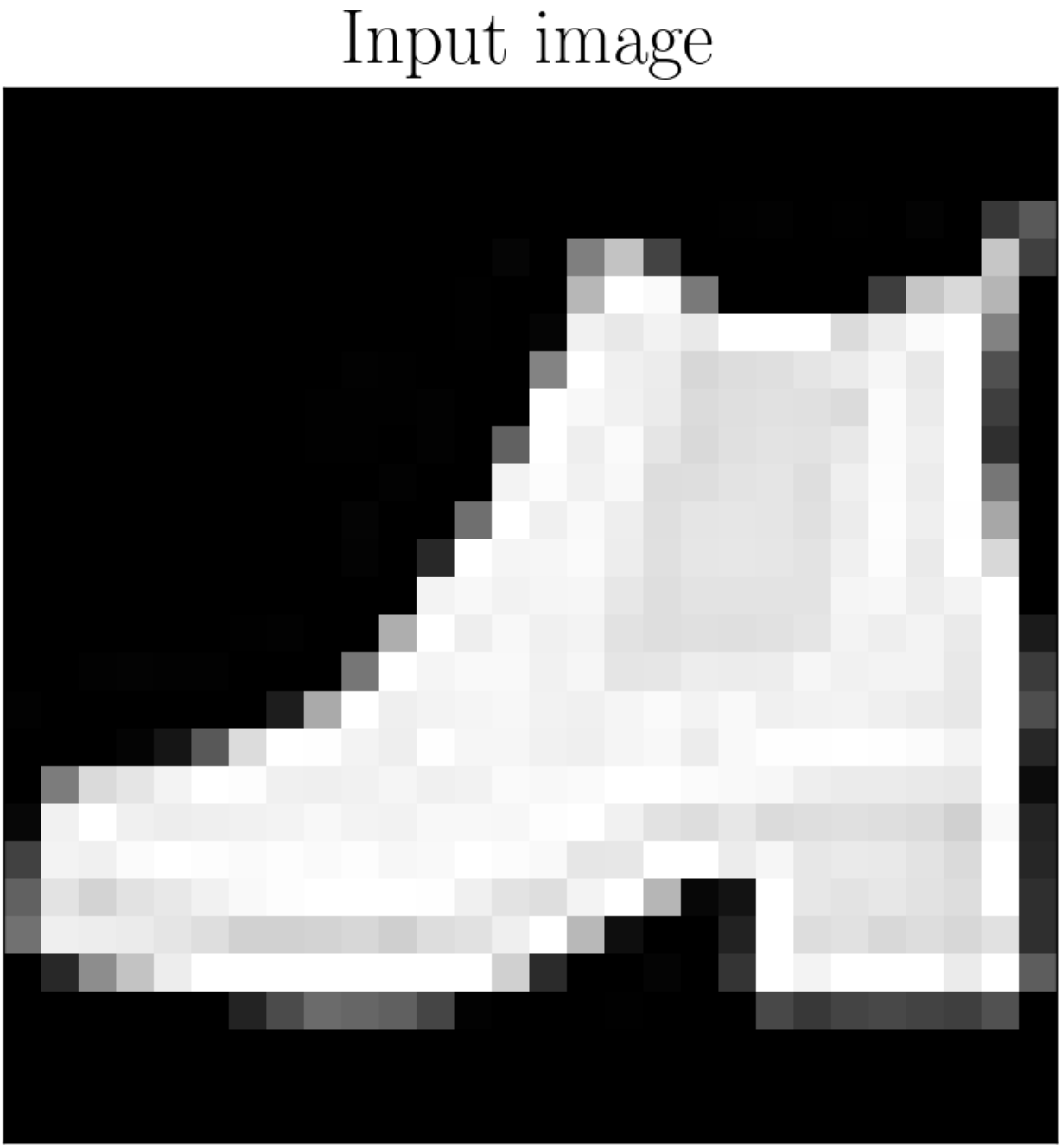
```
with torch.no_grad():
    # every time all the data sets are traversed, test once on the test set
    for img, label in test_data_loader:
        fig = plt.figure(dpi=200)
        plt.imshow(img.squeeze().numpy(), cmap='gray')
        # Note that the size of the image input to the network is ``[1, 1, 28, 28]``,␣
↪the 0th dimension is ``batch``, and the first dimension is ``channel``
        # therefore, when calling ``imshow``, first use ``squeeze()`` to change the␣
↪size to ``[28, 28]``
        plt.title('Input image', fontsize=20)
```

(下页继续)

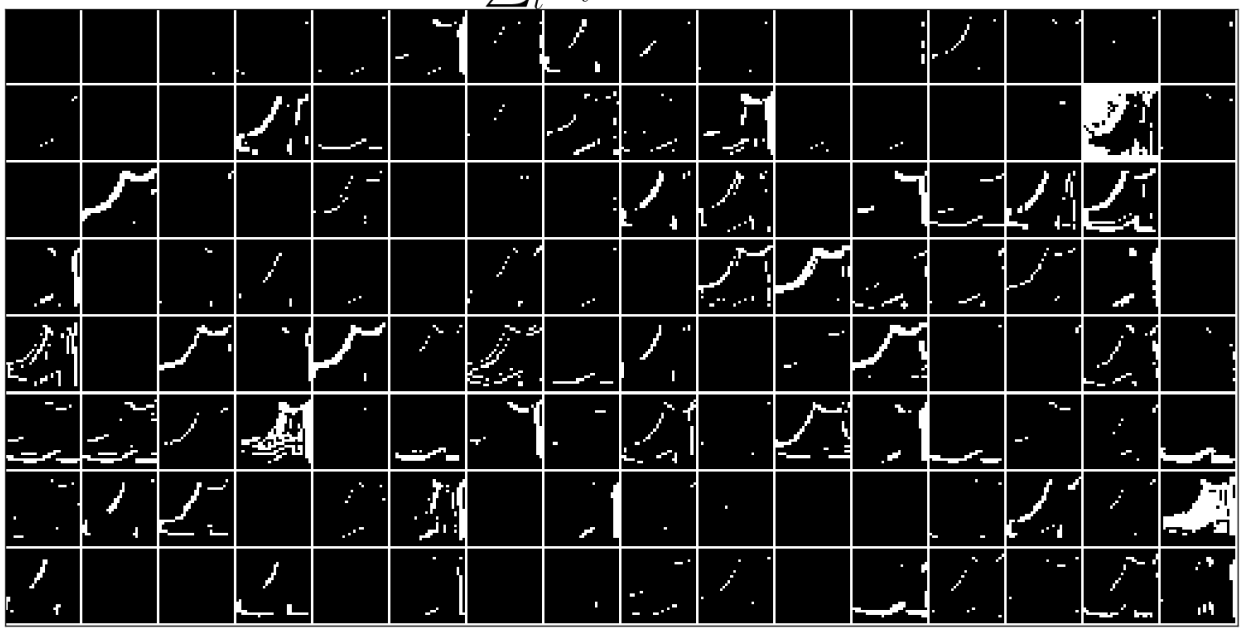
(续上页)

```
        plt.xticks([])
        plt.yticks([])
        plt.show()
        out_spikes = 0
        for t in range(net.T):
            out_spikes += encoder(img).squeeze()
            # the size of encoder(img) is ``[1, 128, 28, 28]``, the same use␣
↪``squeeze()`` transform size to ``[128, 28, 28]``
            if t == 0 or t == net.T - 1:
                out_spikes_c = out_spikes.clone()
                for i in range(out_spikes_c.shape[0]):
                    if out_spikes_c[i].max().item() > out_spikes_c[i].min().item():
                        # Normalize each feature map to make the display clearer
                        out_spikes_c[i] = (out_spikes_c[i] - out_spikes_c[i].min()) /␣
↪(out_spikes_c[i].max() - out_spikes_c[i].min())
                visualizing.plot_2d_spiking_feature_map(out_spikes_c, 8, 16, 1, None)
                plt.title('$\\sum_{t} S_{t}$ at $t = ' + str(t) + '$', fontsize=20)
                plt.show()
```

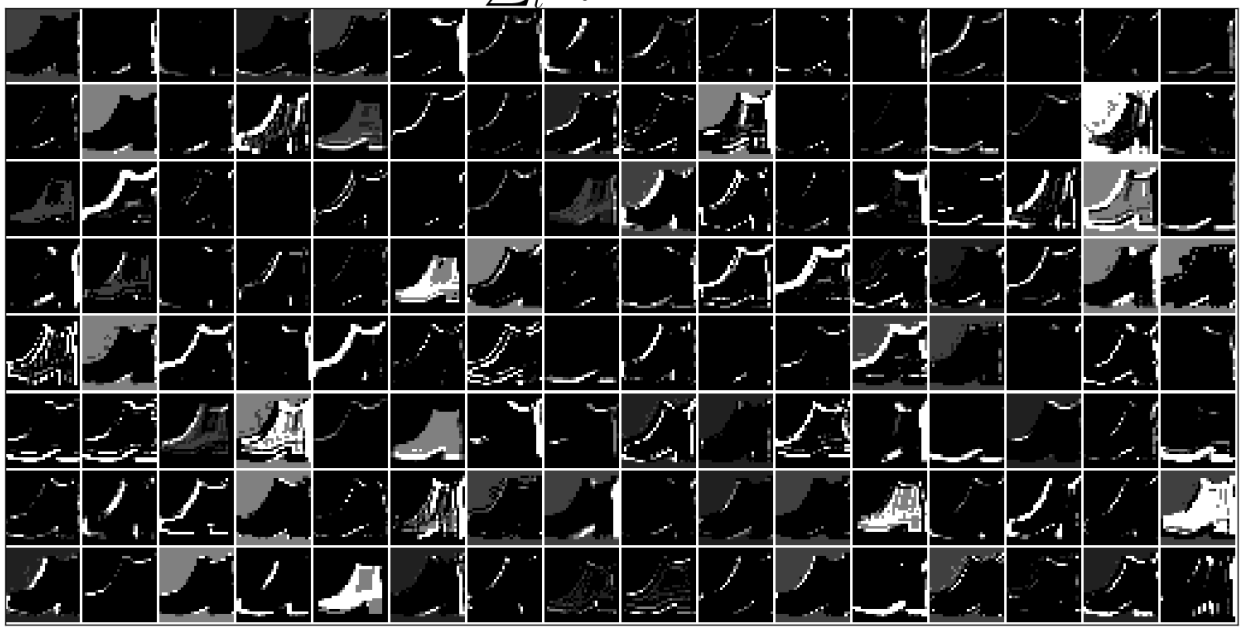
The following shows two input pictures and the cumulative spike $\sum_t S_t$ output by the encoder at the begin time of `t=0` and the end time `t=7`:

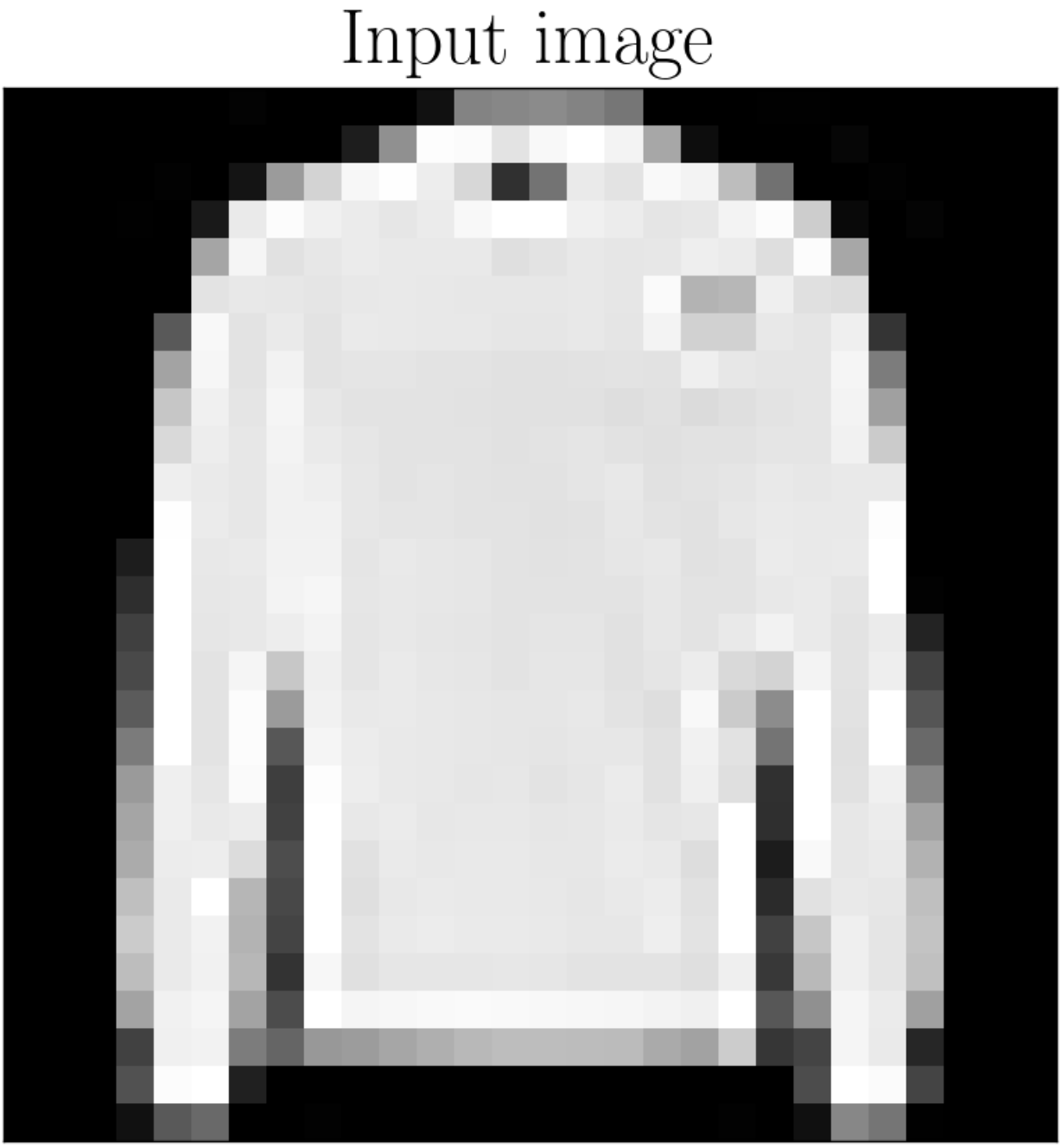

Input image

$\sum_t S_t$ at $t = 0$

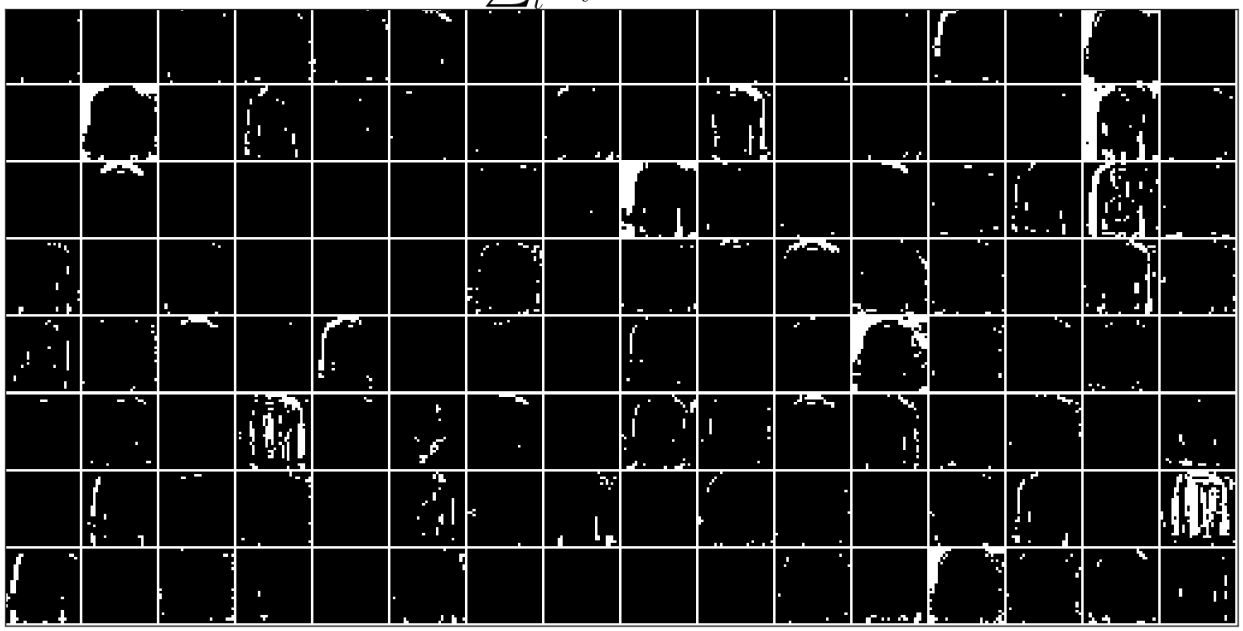


$\sum_t S_t$ at $t = 7$

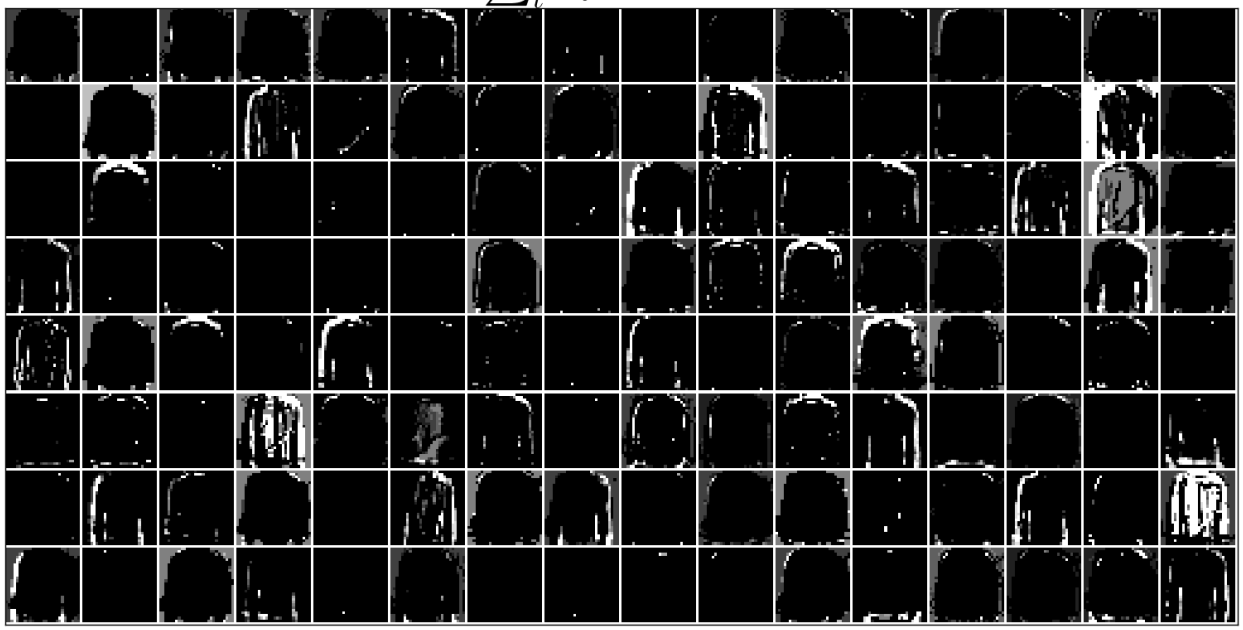



Input image

$\sum_t S_t$ at $t = 0$



$\sum_t S_t$ at $t = 7$



Observation shows that the cumulative output spike $\sum_t S_t$ of the encoder is very close to the contour of the original image. It seems that this kind of self-learning spike encoder has strong coding ability.

6.1.5 spikingjelly.clock_driven.ann2snn

Author: DingJianhao, fangwei123456

This tutorial focuses on `spikingjelly.clock_driven.ann2snn`, introduce how to convert the trained feed-forward ANN to SNN and simulate it on the SpikingJelly framework.

Currently support conversion of Pytorch modules including `nn.Conv2d` , `nn.Linear` , `nn.MaxPool2d` , `nn.AvgPool2d` , `nn.BatchNorm1d` , `nn.BatchNorm2d` , `nn.Flatten` , `nn.ReLU` ,other module solutions are under development…

Theoretical basis of ANN2SNN

Compared with ANN, SNN generates discrete spikes, which is conducive to efficient communication. Today, ANN is popular, while direct training of SNN requires far more resources. Naturally, people will think of using very mature ANN to switch to SNN, and hope that SNN can have similar performance. This leads to the question of how to build a bridge between ANN and SNN. The current SNN mainstream method is to use frequency coding. So for the output layer, we will use the number of neuron output spikes to determine the category. Is the firing rate related to ANN?

Fortunately, there is a strong correlation between the non-linear activation of ReLU neurons in ANN and the firing rate of IF neurons in SNN (reset by subtracting the threshold $V_{threshold}$). We can use this feature for conversion. The neuron update method mentioned here is the Soft method mentioned in the Clock Driven Tutorial.

The following figure shows this correspondence: the left figure is a curve obtained by giving a constant input to an IF neuron and observing its firing over a period of time. The right one is the ReLU activation curve, which satisfies $activation = max(input, 0)$.

The literature[1] provides a theoretical basis for analyzing the conversion of ANN to SNN. The theory shows that the IF neuron in SNN is an unbiased estimator of ReLU activation function over time.

For the first layer of the neural network, the input layer, discuss the relationship between the firing rate of SNN neurons r and the activation in the corresponding ANN. Assume that the input is constant as $z \in [0, 1]$. For the IF neuron reset by subtraction, its membrane potential V changes with time as follows:

$$V_t = V_{t-1} + z - V_{threshold}\theta_t$$

Where: $V_{threshold}$ is the firing threshold, usually set to 1.0. θ_t is the output spike. The average firing rate in the T time steps can be obtained by summing the membrane potential:

$$\sum_{t=1}^{T} V_t = \sum_{t=1}^{T} V_{t-1} + zT - V_{threshold}\sum_{t=1}^{T}\theta_t$$

Move all the items containing V_t to the left, and divide both sides by T:

$$\frac{V_T - V_0}{T} = z - V_{threshold}\frac{\sum_{t=1}^{T}\theta_t}{T} = z - V_{threshold}\frac{N}{T}$$

[1] Rueckauer B, Lungu I-A, Hu Y, Pfeiffer M and Liu S-C (2017) Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification. Front. Neurosci. 11:682.

Where N is the number of pulses in the time step of T, and $\frac{N}{T}$ is the issuing rate r. Use $z = V_{threshold}a$ which is:

$$r = a - \frac{V_T - V_0}{TV_{threshold}}$$

Therefore, when the simulation time step T is infinite:

$$r = a(a > 0)$$

Similarly, for the higher layers of the neural network, literature[?] further explains that the inter-layer firing rate satisfies:

$$r^l = W^l r^{l-1} + b^l - \frac{V_T^l}{TV_{threshold}}$$

For details, please refer to[?]. The methods in ann2snn also mainly come from[?] .

Conversion and simulation

Specifically, there are two main steps for converting feedforward ANN to SNN: model parsing and model simulation.

model parsing

Model parsing mainly solves two problems:

1. Researchers propose Batch Normalization for fast training and convergence. Batch normalization aims to normalize the output of ANN to 0 mean, which is contrary to the characteristics of SNN. Therefore, the parameters of BN need to be absorbed into the previous parameter layer (Linear, Conv2d)
2. According to the conversion theory, the input and output of each layer of ANN need to be limited to the range of [0,1], which requires scaling of the parameters (model normalization)

◆ Absorbing BatchNorm parameters

Assume that the parameters of BatchNorm are γ (BatchNorm.weight), β (BatchNorm.bias), μsigma‘(BatchNorm.running_std, square root of running_var).For specific parameter definitions, see `torch.nn.batchnorm`. Parameter modules (such as Linear) have parameters W and b. Absorbing BatchNorm parameters is transfering the parameters of BatchNorm to W and b of the parameter module through calculation， , so that the output of the data in new module is the same as when there is BatchNorm. In this regard, the new model’ s $\bar{W}$ and $\bar{b}$ formulas are expressed as:

$$\bar{W} = \frac{\gamma}{\sigma} W$$

$$\bar{b} = \frac{\gamma}{\sigma}(b - \mu) + \beta$$

◆ Model normalization

For a parameter module, assuming that the input tensor and output tensor are obtained, the maximum value of the input tensor is λ_{pre}, and the maximum value of the output tensor is λ Then, the normalized weight $\hat{W}$ is:

$$\hat{W} = W * \frac{\lambda_{pre}}{\lambda}$$

The normalized bias $\hat{b}$ is:

$$\hat{b} = b/\lambda$$

Although the output distribution of each layer of ANN obeys a certain distribution, there are often large outliers in the data, which will reduce the overall neuron firing rate. To solve this problem, robust normalization adjusts the scaling factor from the maximum value of the tensor to the p-percentile of the tensor. The recommended percentile value in the literature is 99.9

So far, the operations we have done on neural networks are completely equivalent. The performance of the current model should be the same as the original model.

Model simulation

Before simulation, we need to change the ReLU activation function in the original model into an IF neuron. For the average pooling in ANN, we need to transform it into spatial subsampling. Because IF neuron can be equivalent to ReLU activation function. Adding IF neurons after spatial downsampling has little effect on the results. There is currently no ideal solution for maximum pooling in ANN. The best solution at present is to control the spike channel[?] with a gated function based on the momentum accumulation spike. This is also the default method in ann2snn. There are also literatures proposing to use spatial subsampling to replace Maxpool2d.

In simulation, according to the conversion theory, SNN needs to input a constant analog input. Using a Poisson encoder will bring about a decrease in accuracy. Both Poisson coding and constant input have been implemented, and one can perform different experiments if interested.

Optional configuration

In view of the various optional configurations in the conversion, the `Config` class implemented in `ann2snn.utils` is used to load the default configuration and save the configuration. By loading the default configuration in Config and modifying it, one can set the parameters required when running.

Below are the introductions of the configuration corresponding to different parameters, the feasible input range, and why this configuration is needed.

(1) conf[‘parser’][‘robust_norm’]

Available value：`bool`

Note： when `True`, use robust normalization

(2) conf[‘simulation’][‘reset_to_zero’]

Available value: `None`, floating point

Note: When floating point, voltage of neurons that just fired spikes will be set to :math:`V_{reset}`; when `None`, voltage of neurons that just fired spikes will subtract :math:`V_{threshold}`. For model that need normalization, setting to `None` is default, which has theoretical guaratee.

(3) conf[‘simulation’][‘encoder’][‘possion’]

Available value：`bool`

Note: When `True`, use Possion encoder; otherwise, use constant input over T steps.

(4) conf[‘simulation’][‘avg_pool’][‘has_neuron’]

Available value：`bool`

Note: When `True`, avgpool2d is converted to spatial subsampling with a layer of IF neurons; otherwise, it is only converted to spatial subsampling.

(5) conf[‘simulation’][‘max_pool’][‘if_spatial_avg’]

Available value：`bool`

Note: When `True`,maxpool2d is converted to avgpool2d. As referred in many literatures, this method will cause accuracy degrading.

(6) conf[‘simulation’][‘max_pool’][‘if_wta’]

Available value：`bool`

Note: When `True`, maxpool2d in SNN is identical with maxpool2d in ANN. Using maxpool2d in ANN means that when a spike is available in the Receptive Field, output a spike.

(7) conf[‘simulation’][‘max_pool’][‘momentum’]

Available value: `None`, floating point [0,1]

Note: By default, maxpool2d layer is converted into a gated function controled channel based on momentum cumulative spikes. When set to `None`, the spike is accumulated directly. If set to floating point in the range of [0,1], spike momentum is accumulated.

The default configuration is:

```
default_config =
{
'simulation':
        {
        'reset_to_zero': False,
        'encoder':
                {
                'possion': False
                },
        'avg_pool':
                {
                'has_neuron': True
                },
        'max_pool':
                {
```

(下页继续)

(续上页)

```
                'if_spatial_avg': False,
                'if_wta': False,
                'momentum': None
                }
        },
'parser':
        {
        'robust_norm': True
        }
}
```

MNIST classification

Now, use `ann2snn` to build a simple convolutional network to classify the MNIST dataset.

First define our network structure:

```
class ANN(nn.Module):
        def __init__(self):
                super().__init__()
                self.network = nn.Sequential(
                        nn.Conv2d(1, 32, 3, 1),
                        nn.BatchNorm2d(32, eps=1e-3),
                        nn.ReLU(),
                        nn.AvgPool2d(2, 2),

                        nn.Conv2d(32, 32, 3, 1),
                        nn.BatchNorm2d(32, eps=1e-3),
                        nn.ReLU(),
                        nn.AvgPool2d(2, 2),

                        nn.Conv2d(32, 32, 3, 1),
                        nn.BatchNorm2d(32, eps=1e-3),
                        nn.ReLU(),
                        nn.AvgPool2d(2, 2),

                        nn.Flatten(),
                        nn.Linear(32, 10),
                        nn.ReLU()
                )

        def forward(self,x):
                x = self.network(x)
```

(下页继续)

(续上页)

```
                return x
```

Note: In the defined network, the order of module definition must be consistent with the forward order, otherwise it will affect the automatic analysis of the network.It is best to use `nn.Sequence(·)` to completely define the network. After each Conv2d and Linear layer, a ReLU layer must be placed, which can be separated by a BatchNorm layer. No ReLU is added after the pooling layer. If you encounter a situation where you need to expand the tensor, define a `nn.Flatten` module in the network. In the forward function, you need to use the defined Flatten instead of the view function.

Define our hyperparameters:

```
    device = input('输入运行的设备，例如 “cpu” 或 “cuda:0” \n input device, e.g., "cpu" or
↪"cuda:0": ')
dataset_dir = input('输入保存 MNIST 数据集的位置，例如 “./” \n input root directory for␣
↪saving MNIST dataset, e.g., "./": ')
batch_size = int(input('输入 batch_size，例如 “64” \n input batch_size, e.g., "64": '))
learning_rate = float(input('输入学习率，例如 “1e-3” \n input learning rate, e.g., "1e-3
↪": '))
T = int(input('输入仿真时长，例如 “100” \n input simulating steps, e.g., "100": '))
train_epoch = int(input('输入训练轮数，即遍历训练集的次数，例如 “10” \n input training epochs,
↪ e.g., "10": '))
model_name = input('输入模型名字，例如 “mnist” \n input model name, for log_dir generating␣
↪, e.g., "mnist": ')
```

The program searches for the trained model archive (a file with the same name as *model_name*) according to the specified folder, and all subsequent temporary files will be stored in that folder.

Load the default conversion configuration and save

```
config = utils.Config.default_config
print('ann2snn config:\n\t', config)
utils.Config.store_config(os.path.join(log_dir,'default_config.json'),config)
```

Initialize data loader, network, optimizer, loss function

```
# Initialize the network
ann = ANN().to(device)
# Define loss function
loss_function = nn.CrossEntropyLoss()
# Use Adam optimizer
optimizer = torch.optim.Adam(ann.parameters(), lr=learning_rate, weight_decay=5e-4)
```

Train ANN and test it regularly. You can also use the pre-written training program in utils during training.

```
for epoch in range(train_epoch):
        # Train the network using a pre-prepared code in ''utils''
```

(下页继续)

(续上页)

```
        utils.train_ann(net=ann,
                                    device=device,
                                    data_loader=train_data_loader,
                                    optimizer=optimizer,
                                    loss_function=loss_function,
                                    epoch=epoch
                                    )
        # Validate the network using a pre-prepared code in ''utils''
        acc = utils.val_ann(net=ann,
                                          device=device,
                                          data_loader=test_data_loader,
                                          epoch=epoch
                                          )
        if best_acc <= acc:
                utils.save_model(ann, log_dir, model_name+'.pkl')
```

The complete code is located in `ann2snn.examples.if_cnn_mnist.py`, in the code we also use Tensorboard to save training logs. You can run it directly on the Python command line:

```
>>> import spikingjelly.clock_driven.ann2snn.examples.if_cnn_mnist as if_cnn_mnist
>>> if_cnn_mnist.main()
输入运行的设备，例如“cpu”或“cuda:0”
 input device, e.g., "cpu" or "cuda:0": cuda:15
输入保存 MNIST 数据集的位置，例如“./”
 input root directory for saving MNIST dataset, e.g., "./": ./mnist
输入 batch_size，例如“64”
 input batch_size, e.g., "64": 128
输入学习率，例如“1e-3”
 input learning rate, e.g., "1e-3": 1e-3
输入仿真时长，例如“100”
 input simulating steps, e.g., "100": 100
输入训练轮数，即遍历训练集的次数，例如“10”
 input training epochs, e.g., "10": 10
输入模型名字，用于自动生成日志文档，例如“mnist”
 input model name, for log_dir generating , e.g., "mnist"

If the input of the main function is not a folder with valid files, an automatic log␣
↪file folder is automatically generated.
 Terminal outputs root directory for saving logs, e.g., "./": ./log-mnist1596804385.
↪476601

Epoch 0 [1/937] ANN Training Loss:2.252 Accuracy:0.078
Epoch 0 [101/937] ANN Training Loss:1.424 Accuracy:0.669
```

(下页继续)

(续上页)

```
Epoch 0 [201/937] ANN Training Loss:1.117 Accuracy:0.773
Epoch 0 [301/937] ANN Training Loss:0.953 Accuracy:0.795
Epoch 0 [401/937] ANN Training Loss:0.865 Accuracy:0.788
Epoch 0 [501/937] ANN Training Loss:0.807 Accuracy:0.792
Epoch 0 [601/937] ANN Training Loss:0.764 Accuracy:0.795
Epoch 0 [701/937] ANN Training Loss:0.726 Accuracy:0.834
Epoch 0 [801/937] ANN Training Loss:0.681 Accuracy:0.880
Epoch 0 [901/937] ANN Training Loss:0.641 Accuracy:0.888
Epoch 0 [100/100] ANN Validating Loss:0.328 Accuracy:0.881
Save model to: ./log-mnist1596804385.476601\mnist.pkl
...
Epoch 9 [901/937] ANN Training Loss:0.036 Accuracy:0.990
Epoch 9 [100/100] ANN Validating Loss:0.042 Accuracy:0.988
Save model to: ./log-mnist1596804957.0179427\mnist.pkl
```

In the example, this model is trained for 10 epochs. The changes in the accuracy of the test set during training are as follows:

In the end, the accuracy on test dataset is 98.8%.

Take a part of the data from the training set and use it for the normalization step of the model. Here we take 1/500 of the training data, which is 100 pictures. But it should be noted that the range of the data tensor taken from the dataset is [0, 255], and it needs to be divided by 255 to become a floating point tensor in the range of [0.0, 1.0] to match the feasible range of firing rate.

```
    norm_set_len = int(train_data_dataset.data.shape[0] / 500)
print('Using %d pictures as norm set'%(norm_set_len))
norm_set = train_data_dataset.data[:norm_set_len, :, :].float() / 255
norm_tensor = torch.FloatTensor(norm_set).view(-1,1,28,28)
```

Call the standard conversion function `standard_conversion` implemented in `ann2snn.utils` to realize ANN conversion and SNN simulation.

```
utils.standard_conversion(model_name=model_name,
                      norm_data=norm_tensor,
                      test_data_loader=test_data_loader,
                      device=device,
                      T=T,
                      log_dir=log_dir,
                      config=config
                      )
```

In the process, the normalized model structure is output:

```
ModelParser(
  (network): Sequential(
        (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
        (1): ReLU()
        (2): AvgPool2d(kernel_size=2, stride=2, padding=0)
        (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
        (4): ReLU()
        (5): AvgPool2d(kernel_size=2, stride=2, padding=0)
        (6): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
        (7): ReLU()
        (8): AvgPool2d(kernel_size=2, stride=2, padding=0)
        (9): Flatten()
        (10): Linear(in_features=32, out_features=10, bias=True)
        (11): ReLU()
  )
)
```

At the same time, one can also observe the structure of SNN:

```
SNN(
  (network): Sequential(
        (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
        (1): IFNode(
          v_threshold=1.0, v_reset=None
          (surrogate_function): Sigmoid()
        )
        (2): AvgPool2d(kernel_size=2, stride=2, padding=0)
        (3): IFNode(
          v_threshold=1.0, v_reset=None
          (surrogate_function): Sigmoid()
        )
        (4): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
        (5): IFNode(
          v_threshold=1.0, v_reset=None
          (surrogate_function): Sigmoid()
        )
        (6): AvgPool2d(kernel_size=2, stride=2, padding=0)
        (7): IFNode(
          v_threshold=1.0, v_reset=None
          (surrogate_function): Sigmoid()
        )
        (8): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
        (9): IFNode(
          v_threshold=1.0, v_reset=None
```

(下页继续)

(续上页)

```
          (surrogate_function): Sigmoid()
        )
        (10): AvgPool2d(kernel_size=2, stride=2, padding=0)
        (11): IFNode(
          v_threshold=1.0, v_reset=None
          (surrogate_function): Sigmoid()
        )
        (12): Flatten()
        (13): Linear(in_features=32, out_features=10, bias=True)
        (14): IFNode(
          v_threshold=1.0, v_reset=None
          (surrogate_function): Sigmoid()
        )
  )
)
```

It can be seen that the activation of ReLU in the ANN model is replaced by the IFNode of SNN. Each layer of AvgPool2d is followed by a layer of IFNode.

Due to the long time of model simulation, the current accuracy and simulation progress are continuously output:

```
[SNN Simulating... 1.00%] Acc:0.990
[SNN Simulating... 2.00%] Acc:0.990
[SNN Simulating... 3.00%] Acc:0.990
[SNN Simulating... 4.00%] Acc:0.988
[SNN Simulating... 5.00%] Acc:0.990
……
[SNN Simulating... 95.00%] Acc:0.986
[SNN Simulating... 96.00%] Acc:0.986
[SNN Simulating... 97.00%] Acc:0.986
[SNN Simulating... 98.00%] Acc:0.986
[SNN Simulating... 99.00%] Acc:0.987
SNN Simulating Accuracy:0.987
Summary:        ANN Accuracy:98.7900%   SNN Accuracy:98.6500% [Decreased 0.1400%]
```

Through the final output, we can know that the accuracy of ANN’s MNIST classification is 98.79%. The accuracy of the converted SNN is 98.65%. The conversion resulted in a 0.14% performance degradation.

6.1.6 Reinforcement Learning: Deep Q Learning

Authors: fangwei123456， lucifer2859

Translator: LiutaoYu

This tutorial applies a spiking neural network to reproduce the PyTorch official tutorial REINFORCEMENT LEARNING (DQN) TUTORIAL. Please make sure that you have read the original tutorial and corresponding codes before proceeding.

Change the input

In the ANN version, the difference between two adjacent frames of CartPole is directly used as input, and then CNN is used to extract features. We can also use the same method for the SNN version. However, to obtain the frames, the graphical interface must be activated, which is not convenient for training on a remote server without a graphical interface. To reduce the difficulty, we directly use CartPole’s state variables as the network input, which is an array containing 4 floating numbers, i.e., *Cart Position*, *Cart Velocity*, *Pole Angle* and *Pole Velocity At Tip*. The training code also needs to be changed accordingly, which will be shown below.

Next, we need to define the SNN structure. Usually in Deep Q Learning, the neural network acts as the Q function, the output of which should be continuous values. This means that the last layer of the SNN should not output spikes representing Q function as 0 and 1, which may lead to poor performance. There are several methods to making SNN output continuous values. For the classification tasks in the previous tutorials, the final output of the network is the firing rate of each neuron in the output layer, which is obtained by counting the number of spikes in the simulation duration and then dividing the number by the duration. Through preliminary testing, we found that using firing rate as Q function can not lead to satisfying performance. Because after simulating T steps, the possible firing rates are $0, \frac{1}{T}, \frac{2}{T}, ..., 1$, which are not enough to represent the Q function.

Here, we apply a new method to make SNN output floating numbers. We set the firing threshold of a neuron to be infinity, which won’t fire at all, and we adopt the final membrane potential to represent Q function. It is convenient to implement such neurons in the `SpikingJelly` framework: just inherit everything from LIF neuron `neuron.LIFNode` and rewrite its `forward` function.

```
class NonSpikingLIFNode(neuron.LIFNode):
    def forward(self, dv: torch.Tensor):
        self.neuronal_charge(dv)
        # self.neuronal_fire()
        # self.neuronal_reset()
        return self.v
```

The structure of the Deep Q Spiking Network is very simple: input layer, IF neuron layer, and NonSpikingLIF neuron layer, between which are fully linear connections. The IF neuron layer is an encoder to convert the CartPole’s state variables to spikes, and the NonSpikingLIF neuron layer can be regraded as the decision making unit.

```
class DQSN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, T=16):
```

(下页继续)

(续上页)

```
        super().__init__()
        self.fc = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            neuron.IFNode(),
            nn.Linear(hidden_size, output_size),
            NonSpikingLIFNode(tau=2.0)
        )

        self.T = T

    def forward(self, x):
        for t in range(self.T):
            self.fc(x)

        return self.fc[-1].v
```

Training the network

The code of this part is almost the same with the ANN version. But note that the SNN version here adopts `Observation` returned by `env` as the input.

Following is the training code of the ANN version:

```
for i_episode in range(num_episodes):
    # Initialize the environment and state
    env.reset()
    last_screen = get_screen()
    current_screen = get_screen()
    state = current_screen - last_screen
    for t in count():
        # Select and perform an action
        action = select_action(state)
        _, reward, done, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device)

        # Observe new state
        last_screen = current_screen
        current_screen = get_screen()
        if not done:
            next_state = current_screen - last_screen
        else:
            next_state = None
```

(下页继续)

(续上页)

```
        # Store the transition in memory
        memory.push(state, action, next_state, reward)

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the target network)
        optimize_model()
        if done:
            episode_durations.append(t + 1)
            plot_durations()
            break
    # Update the target network, copying all weights and biases in DQN
    if i_episode % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict())
```

Here is training code of the SNN version. During the training process, we will save the model parameters responsible for the largest reward.

```
for i_episode in range(num_episodes):
    # Initialize the environment and state
    env.reset()
    state = torch.zeros([1, n_states], dtype=torch.float, device=device)

    total_reward = 0

    for t in count():
        action = select_action(state, steps_done)
        steps_done += 1
        next_state, reward, done, _ = env.step(action.item())
        total_reward += reward
        next_state = torch.from_numpy(next_state).float().to(device).unsqueeze(0)
        reward = torch.tensor([reward], device=device)

        if done:
            next_state = None

        memory.push(state, action, next_state, reward)

        state = next_state
        if done and total_reward > max_reward:
            max_reward = total_reward
            torch.save(policy_net.state_dict(), max_pt_path)
```

(下页继续)

(续上页)

```
            print(f'max_reward={max_reward}, save models')

        optimize_model()

        if done:
            print(f'Episode: {i_episode}, Reward: {total_reward}')
            writer.add_scalar('Spiking-DQN-state-' + env_name + '/Reward', total_
↪reward, i_episode)
            break

    if i_episode % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict())
```

It should be emphasized here that, we need to `reset` the network after each forward process, because SNN is retentive while each trial should be started with a clean network state.

```
def select_action(state, steps_done):
    ...
    if sample > eps_threshold:
        with torch.no_grad():
            ac = policy_net(state).max(1)[1].view(1, 1)
            functional.reset_net(policy_net)
    ...

def optimize_model():
    ...
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0].
↪detach()
    functional.reset_net(target_net)
    ...
    optimizer.step()
    functional.reset_net(policy_net)
```

The integrated script can be found here clock_driven/examples/Spiking_DQN_state.py. And we can start the training process in a Python Console as follows.

```
>>> from spikingjelly.clock_driven.examples import Spiking_DQN_state
>>> Spiking_DQN_state.train(use_cuda=False, model_dir='./model/CartPole-v0', log_dir=
↪'./log', env_name='CartPole-v0', hidden_size=256, num_episodes=500, seed=1)
...
Episode: 509, Reward: 715
```

(下页继续)

(续上页)

```
Episode: 510, Reward: 3051
Episode: 511, Reward: 571
complete
state_dict path is./ policy_net_256.pt
```

Testing the network

After training for 512 episodes, we download the model `policy_net_256_max.pt` that maximizes the reward during the training process from the server, and run the `play` function on a local machine with a graphical interface to test its performance.

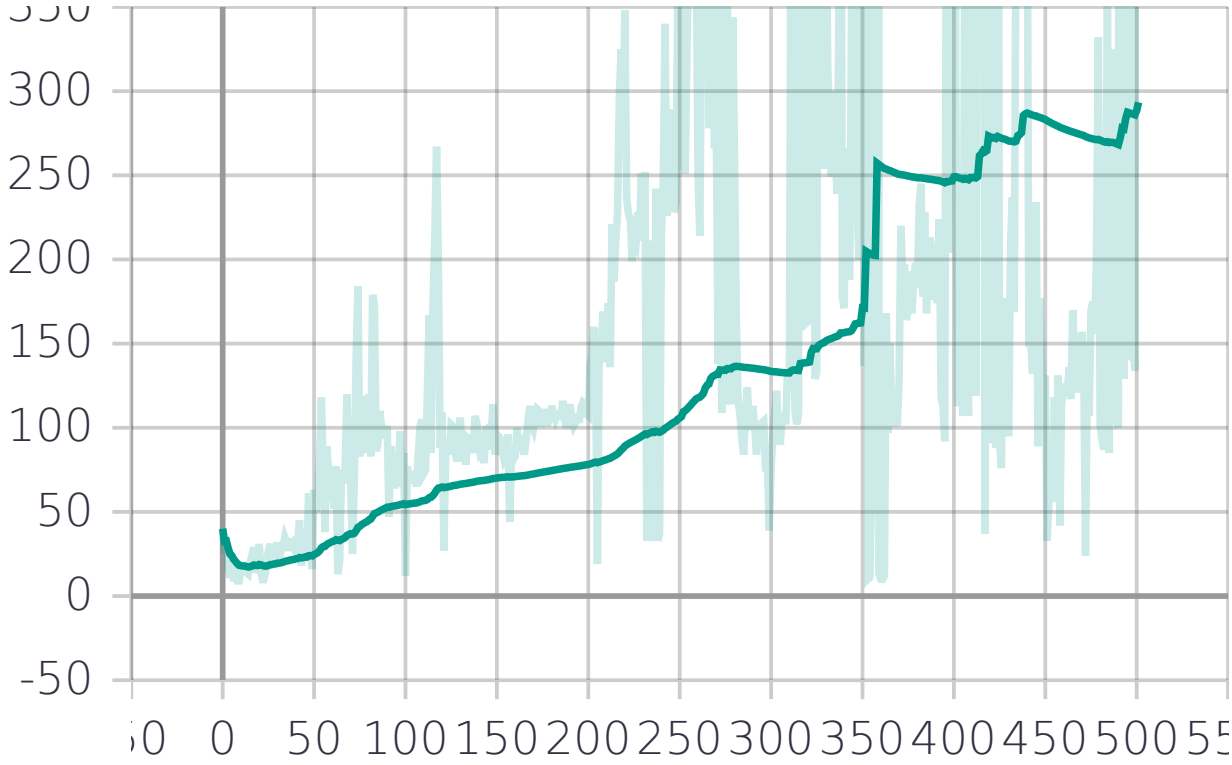
```
>>> from spikingjelly.clock_driven.examples import Spiking_DQN_state
>>> Spiking_DQN_state.play(use_cuda=False, pt_path='./model/CartPole-v0/policy_net_
↪256_max.pt', env_name='CartPole-v0', hidden_size=256, played_frames=300)
```

The trained SNN controls the left or right movement of the CartPole, until the end of the game or the number of continuous frames exceeds `played_frames`. During the simulation, the `play` function will draw the firing rate of the IF neuron, and the voltages of the NonSpikingLIF neurons in the output layer at the last moment, which directly determine the movement of the CartPole.

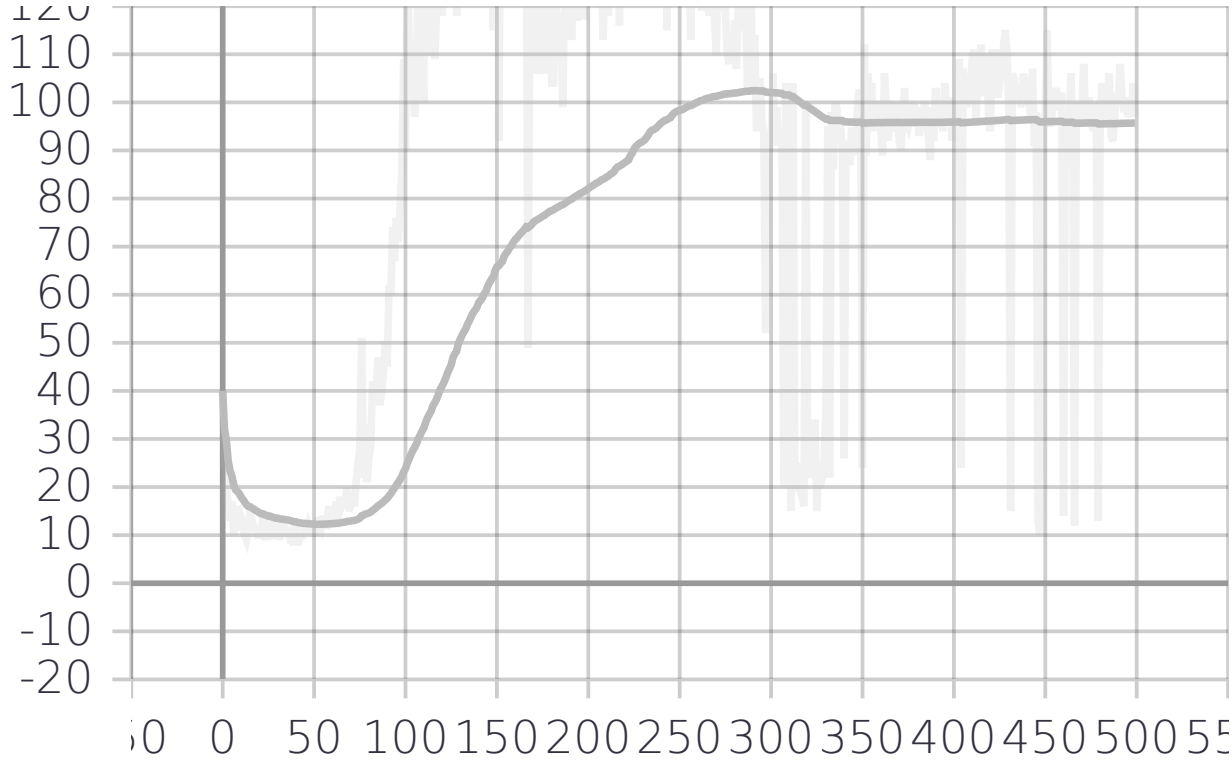
The performance after 16 episodes:

The performance after 32 episodes:

The reward increases with training:



Here is the performance of the ANN version (The code can be found here clock_driven/examples/DQN_state.py).



6.1.7 Reinforcement Learning: Advantage Actor Critic (A2C)

Author: lucifer2859

Translator: LiutaoYu

This tutorial applies a spiking neural network to reproduce actor-critic.py. Please make sure that you have read the original tutorial and corresponding codes before proceeding.

Here, we apply the same method as the previous DQN tutorial to make SNN output floating numbers. We set the firing threshold of a neuron to be infinity, which won't fire at all, and we adopt the final membrane potential to represent Q function. It is convenient to implement such neurons in the `SpikingJelly` framework: just inherit everything from LIF neuron `neuron.LIFNode` and rewrite its `forward` function.

```
class NonSpikingLIFNode(neuron.LIFNode):
def forward(self, dv: torch.Tensor):
    self.neuronal_charge(dv)
    # self.neuronal_fire()
    # self.neuronal_reset()
    return self.v
```

The basic structure of the Spiking Actor-Critic Network is very simple: input layer, IF neuron layer, and NonSpikingLIF neuron layer, between which are fully linear connections. The IF neuron layer is an encoder to convert the CartPole's state variables to spikes, and the NonSpikingLIF neuron layer can be regraded as the decision making unit.

```
class ActorCritic(nn.Module):
    def __init__(self, num_inputs, num_outputs, hidden_size, T=16):
        super(ActorCritic, self).__init__()

        self.critic = nn.Sequential(
            nn.Linear(num_inputs, hidden_size),
            neuron.IFNode(),
            nn.Linear(hidden_size, 1),
            NonSpikingLIFNode(tau=2.0)
        )

        self.actor = nn.Sequential(
            nn.Linear(num_inputs, hidden_size),
            neuron.IFNode(),
            nn.Linear(hidden_size, num_outputs),
            NonSpikingLIFNode(tau=2.0)
        )

        self.T = T

    def forward(self, x):
```

(下页继续)

(续上页)

```
        for t in range(self.T):
            self.critic(x)
            self.actor(x)
        value = self.critic[-1].v
        probs = F.softmax(self.actor[-1].v, dim=1)
        dist  = Categorical(probs)

        return dist, value
```

Training the network

The code of this part is almost the same with the ANN version. But note that the SNN version here adopts `Observation` returned by `env` as the network input.

Following is the training code of the SNN version. During the training process, we will save the model parameters responsible for the largest reward.

```
while step_idx < max_steps:

    log_probs = []
    values    = []
    rewards   = []
    masks     = []
    entropy = 0

    for _ in range(num_steps):
        state = torch.FloatTensor(state).to(device)
        dist, value = model(state)
        functional.reset_net(model)

        action = dist.sample()
        next_state, reward, done, _ = envs.step(action.cpu().numpy())

        log_prob = dist.log_prob(action)
        entropy += dist.entropy().mean()

        log_probs.append(log_prob)
        values.append(value)
        rewards.append(torch.FloatTensor(reward).unsqueeze(1).to(device))
        masks.append(torch.FloatTensor(1 - done).unsqueeze(1).to(device))

        state = next_state
        step_idx += 1
```

(下页继续)

(续上页)

```
        if step_idx % 1000 == 0:
            test_reward = test_env()
            print('Step: %d, Reward: %.2f' % (step_idx, test_reward))
            writer.add_scalar('Spiking-A2C-multi_env-' + env_name + '/Reward', test_
↪reward, step_idx)

    next_state = torch.FloatTensor(next_state).to(device)
    _, next_value = model(next_state)
    functional.reset_net(model)
    returns = compute_returns(next_value, rewards, masks)

    log_probs = torch.cat(log_probs)
    returns   = torch.cat(returns).detach()
    values    = torch.cat(values)

    advantage = returns - values

    actor_loss  = - (log_probs * advantage.detach()).mean()
    critic_loss = advantage.pow(2).mean()

    loss = actor_loss + 0.5 * critic_loss - 0.001 * entropy

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

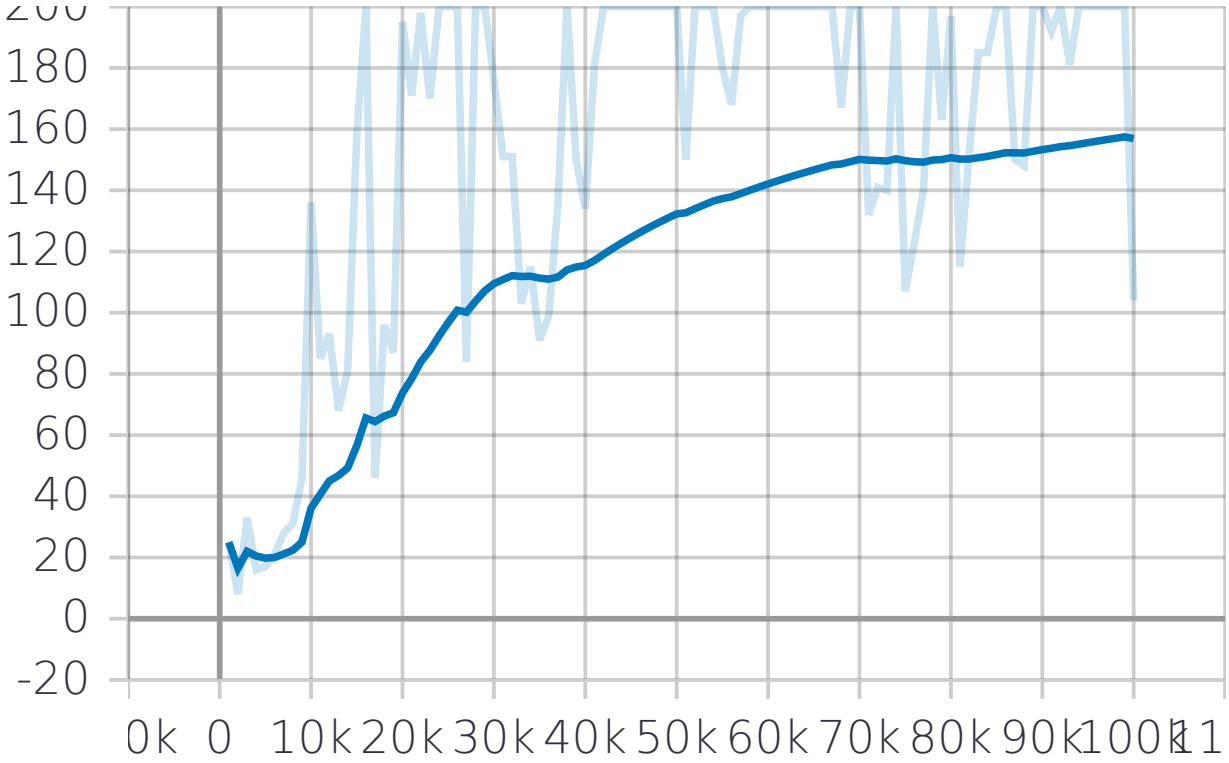
It should be emphasized here that, we need to `reset` the network after each forward process, because SNN is retentive while each trial should be started with a clean network state.

The integrated script can be found here clock_driven/examples/Spiking_A2C.py. And we can start the training process in a Python Console as follows.

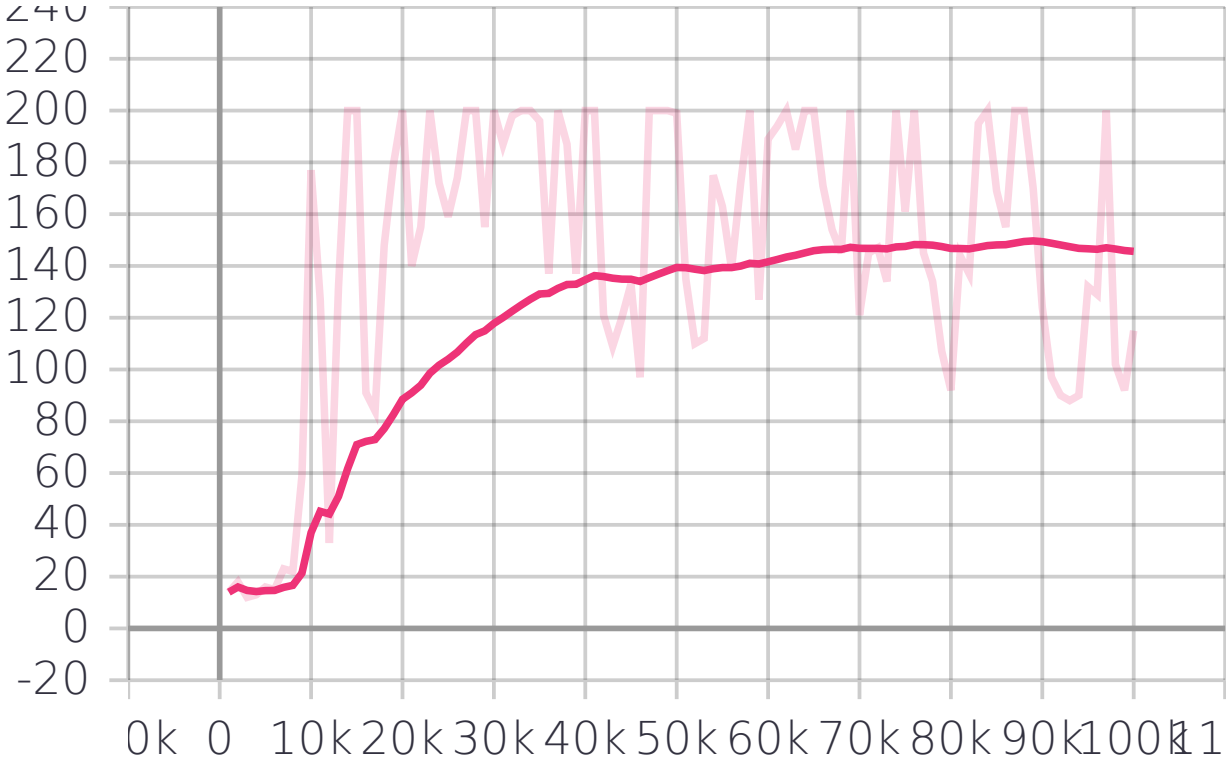
```
>>> python Spiking_A2C.py
```

Performance comparison between ANN and SNN

Here is the reward curve during the training process of 1e5 episodes:



And here is the result of the ANN version with the same settings. The integrated code can be found here clock_driven/examples/A2C.py.



6.1.8 Reinforcement Learning: Proximal Policy Optimization (PPO)

Author: lucifer2859

Translator: LiutaoYu

This tutorial applies a spiking neural network to reproduce ppo.py. Please make sure that you have read the original tutorial and corresponding codes before proceeding.

Here, we apply the same method as the previous DQN tutorial to make SNN output floating numbers. We set the firing threshold of a neuron to be infinity, which won't fire at all, and we adopt the final membrane potential to represent Q function. It is convenient to implement such neurons in the `SpikingJelly` framework: just inherit everything from LIF neuron `neuron.LIFNode` and rewrite the `forward` function.

```
class NonSpikingLIFNode(neuron.LIFNode):
    def forward(self, dv: torch.Tensor):
        self.neuronal_charge(dv)
        # self.neuronal_fire()
        # self.neuronal_reset()
        return self.v
```

The basic structure of the Spiking Actor-Critic Network is very simple: input layer, IF neuron layer, and NonSpikingLIF neuron layer, between which are fully linear connections. The IF neuron layer is an encoder to convert the CartPole's state variables to spikes, and the NonSpikingLIF neuron layer can be regraded as the decision making unit.

```
class ActorCritic(nn.Module):
    def __init__(self, num_inputs, num_outputs, hidden_size, T=16, std=0.0):
        super(ActorCritic, self).__init__()

        self.critic = nn.Sequential(
            nn.Linear(num_inputs, hidden_size),
            neuron.IFNode(),
            nn.Linear(hidden_size, 1),
            NonSpikingLIFNode(tau=2.0)
        )

        self.actor = nn.Sequential(
            nn.Linear(num_inputs, hidden_size),
            neuron.IFNode(),
            nn.Linear(hidden_size, num_outputs),
            NonSpikingLIFNode(tau=2.0)
        )

        self.log_std = nn.Parameter(torch.ones(1, num_outputs) * std)

        self.T = T

    def forward(self, x):
        for t in range(self.T):
            self.critic(x)
            self.actor(x)
        value = self.critic[-1].v
        mu    = self.actor[-1].v
        std   = self.log_std.exp().expand_as(mu)
        dist  = Normal(mu, std)
        return dist, value
```

Training the network

The code of this part is almost the same with the ANN version. But note that the SNN version here adopts `Observation` returned by `env` as the network input.

Following is the training code of the SNN version. During the training process, we will save the model parameters responsible for the largest reward.

```
# GAE
def compute_gae(next_value, rewards, masks, values, gamma=0.99, tau=0.95):
    values = values + [next_value]
    gae = 0
```

(下页继续)

(续上页)

```
    returns = []
    for step in reversed(range(len(rewards))):
        delta = rewards[step] + gamma * values[step + 1] * masks[step] - values[step]
        gae = delta + gamma * tau * masks[step] * gae
        returns.insert(0, gae + values[step])
    return returns

# Proximal Policy Optimization Algorithm
# Arxiv: "https://arxiv.org/abs/1707.06347"
def ppo_iter(mini_batch_size, states, actions, log_probs, returns, advantage):
    batch_size = states.size(0)
    ids = np.random.permutation(batch_size)
    ids = np.split(ids[:batch_size // mini_batch_size * mini_batch_size], batch_size /
↪/ mini_batch_size)
    for i in range(len(ids)):
        yield states[ids[i], :], actions[ids[i], :], log_probs[ids[i], :],␣
↪returns[ids[i], :], advantage[ids[i], :]

def ppo_update(ppo_epochs, mini_batch_size, states, actions, log_probs, returns,␣
↪advantages, clip_param=0.2):
    for _ in range(ppo_epochs):
        for state, action, old_log_probs, return_, advantage in ppo_iter(mini_batch_
↪size, states, actions, log_probs, returns, advantages):
            dist, value = model(state)
            functional.reset_net(model)
            entropy = dist.entropy().mean()
            new_log_probs = dist.log_prob(action)

            ratio = (new_log_probs - old_log_probs).exp()
            surr1 = ratio * advantage
            surr2 = torch.clamp(ratio, 1.0 - clip_param, 1.0 + clip_param) * advantage

            actor_loss  = - torch.min(surr1, surr2).mean()
            critic_loss = (return_ - value).pow(2).mean()

            loss = 0.5 * critic_loss + actor_loss - 0.001 * entropy

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

while step_idx < max_steps:
```

(下页继续)

(续上页)

```
    log_probs = []
    values    = []
    states    = []
    actions   = []
    rewards   = []
    masks     = []
    entropy = 0

    for _ in range(num_steps):
        state = torch.FloatTensor(state).to(device)
        dist, value = model(state)
        functional.reset_net(model)

        action = dist.sample()
        next_state, reward, done, _ = envs.step(torch.max(action, 1)[1].cpu().numpy())

        log_prob = dist.log_prob(action)
        entropy += dist.entropy().mean()

        log_probs.append(log_prob)
        values.append(value)
        rewards.append(torch.FloatTensor(reward).unsqueeze(1).to(device))
        masks.append(torch.FloatTensor(1 - done).unsqueeze(1).to(device))

        states.append(state)
        actions.append(action)

        state = next_state
        step_idx += 1

        if step_idx % 100 == 0:
            test_reward = test_env()
            print('Step: %d, Reward: %.2f' % (step_idx, test_reward))
            writer.add_scalar('Spiking-PPO-' + env_name + '/Reward', test_reward,␣
↪step_idx)

    next_state = torch.FloatTensor(next_state).to(device)
    _, next_value = model(next_state)
    functional.reset_net(model)
    returns = compute_gae(next_value, rewards, masks, values)

    returns   = torch.cat(returns).detach()
    log_probs = torch.cat(log_probs).detach()
```

(下页继续)

(续上页)

```
    values     = torch.cat(values).detach()
    states     = torch.cat(states)
    actions    = torch.cat(actions)
    advantage = returns - values

    ppo_update(ppo_epochs, mini_batch_size, states, actions, log_probs, returns,␣
↪advantage)
```

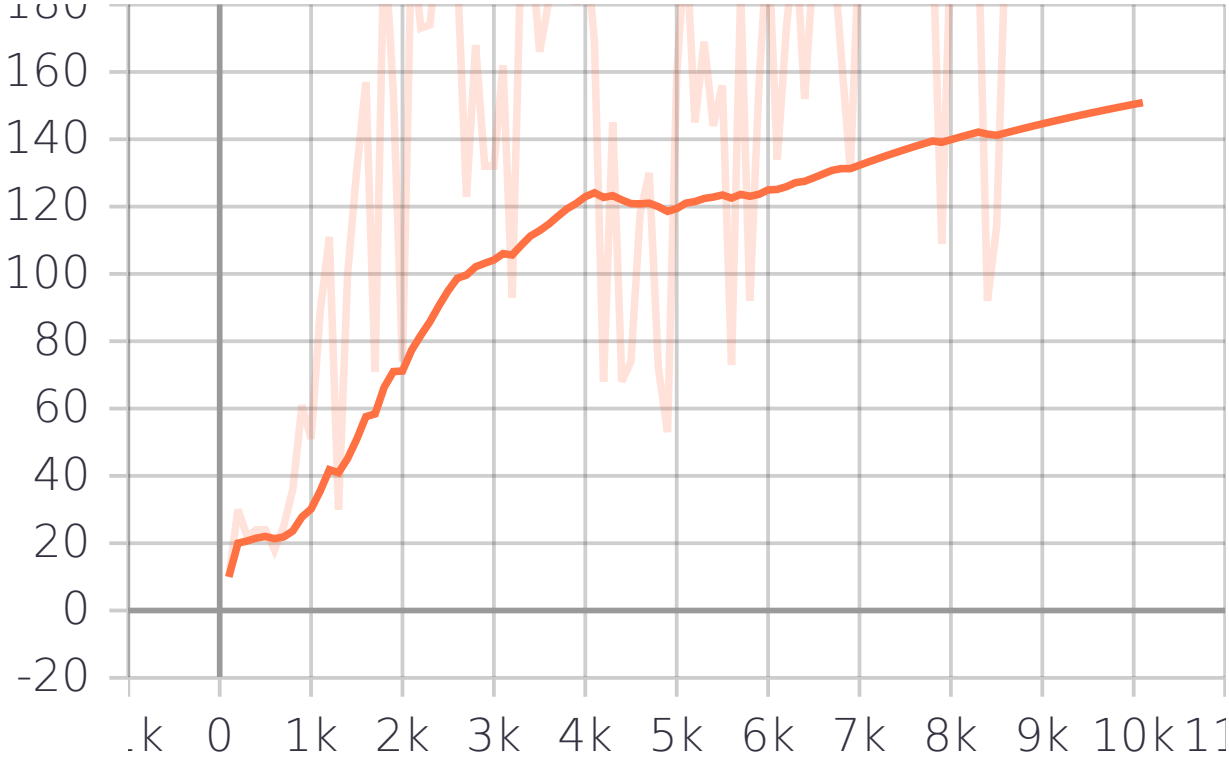
It should be emphasized here that, we need to `reset` the network after each forward process, because SNN is retentive while each trial should be started with a clean network state.

The integrated script can be found here clock_driven/examples/Spiking_PPO.py. And we can start the training process in a Python Console as follows.

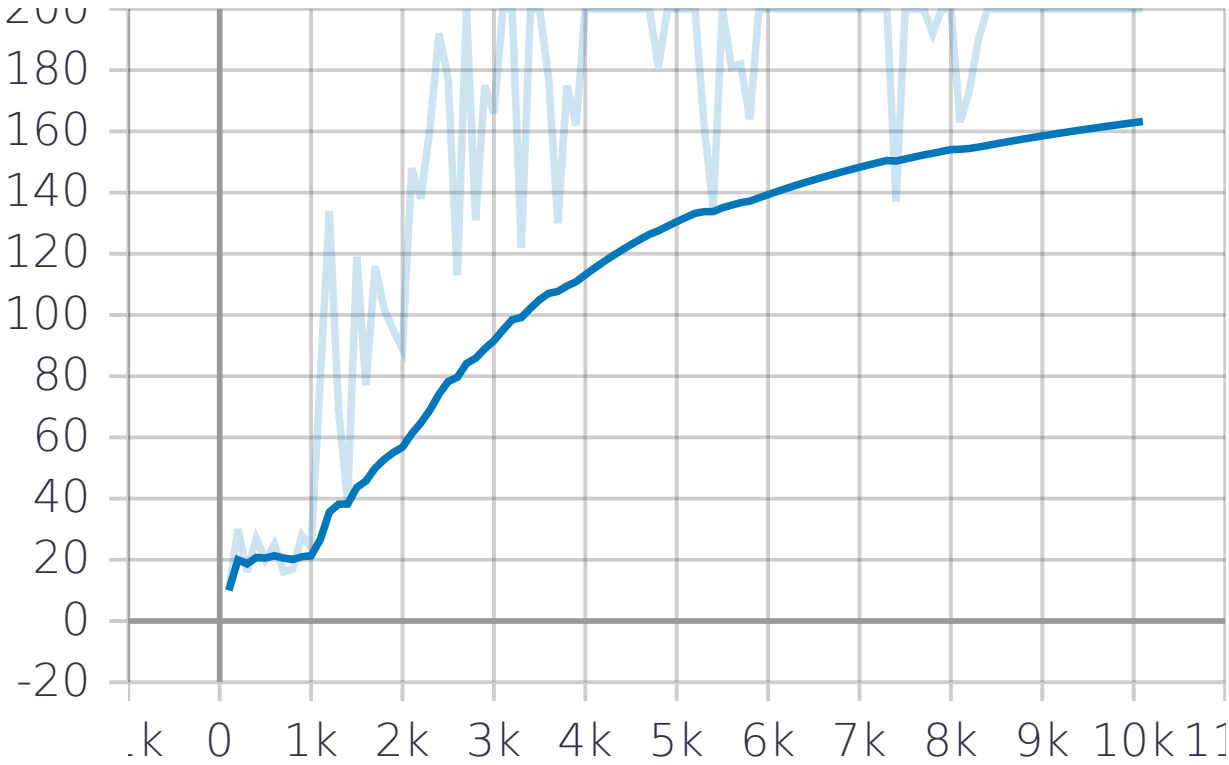
```
>>> python Spiking_PPO.py
```

Performance comparison between ANN and SNN

Here is the reward curve during the training process of 1e5 episodes:



And here is the result of the ANN version with the same settings. The integrated code can be found here clock_driven/examples/PPO.py.



6.1.9 Classifying Names with a Character-level Spiking LSTM

Authors: LiutaoYu, fangwei123456

This tutorial applies a Spiking LSTM to reproduce the PyTorch official tutorial NLP From Scratch: Classifying Names with a Character-Level RNN. Please make sure that you have read the original tutorial and corresponding codes before proceeding. Specifically, we will train a spiking LSTM to classify surnames into different languages according to their spelling, based on a dataset consisting of several thousands of surnames from 18 languages of origin. The integrated script can be found here (clock_driven/examples/spiking_lstm_text.py).

Preparing the data

First of all, we need to download and preprocess the data as the original tutorial, which produces a dictionary `{language: [names ...]}` . Then, we split the dataset into a training set and a testing set (the ratio is 4:1), i.e., `category_lines_train` and `category_lines_test` . Here, we emphasize several important variables: `all_categories` is the list of 18 languages, the length of which is `n_categories=18`; `n_letters=58` is the number of all characters composing the surnames.

```
# split the data into training set and testing set
numExamplesPerCategory = []
category_lines_train = {}
category_lines_test = {}
```

(下页继续)

(续上页)

```
testNumtot = 0
for c, names in category_lines.items():
    category_lines_train[c] = names[:int(len(names)*0.8)]
    category_lines_test[c] = names[int(len(names)*0.8):]
    numExamplesPerCategory.append([len(category_lines[c]), len(category_lines_
↪train[c]), len(category_lines_test[c])])
    testNumtot += len(category_lines_test[c])
```

In addition, we rephrase the function `randomTrainingExample()` to function `randomPair(sampleSource)` for different conditions. Here we adopt function `lineToTensor()` and `randomChoice()` from the original tutorial. `lineToTensor()` converts a surname into a one-hot tensor, and `randomChoice()` randomly choose a sample from the dataset.

```
# Preparing [x, y] pair
def randomPair(sampleSource):
    """
    Args:
        sampleSource:  'train', 'test', 'all'
    Returns:
        category, line, category_tensor, line_tensor
    """
    category = randomChoice(all_categories)
    if sampleSource == 'train':
        line = randomChoice(category_lines_train[category])
    elif sampleSource == 'test':
        line = randomChoice(category_lines_test[category])
    elif sampleSource == 'all':
        line = randomChoice(category_lines[category])
    category_tensor = torch.tensor([all_categories.index(category)], dtype=torch.
↪float)
    line_tensor = lineToTensor(line)
    return category, line, category_tensor, line_tensor
```

Building a spiking LSTM network

We build a spiking LSTM based on the `rnn` module from spikingjelly . The theory can be found in the paper Long Short-Term Memory Spiking Networks and Their Applications . The amounts of neurons in the input layer, hidden layer and output layer are `n_letters`, `n_hidden` and `n_categories` respectively. We add a fully connected layer to the output layer, and use `softmax` function to obtain the classification probability.

```
from spikingjelly.clock_driven import rnn
n_hidden = 256
```

(下页继续)

(续上页)

```
class Net(nn.Module):
    def __init__(self, n_letters, n_hidden, n_categories):
        super().__init__()
        self.n_input = n_letters
        self.n_hidden = n_hidden
        self.n_out = n_categories
        self.lstm = rnn.SpikingLSTM(self.n_input, self.n_hidden, 1)
        self.fc = nn.Linear(self.n_hidden, self.n_out)

    def forward(self, x):
        x, _ = self.lstm(x)
        output = self.fc(x[-1])
        output = F.softmax(output, dim=1)
        return output
```

Training the network

First of all, we initialize the `net` , and define parameters like `TRAIN_EPISODES` and `learning_rate`. Here we adopt `mse_loss` and `Adam` optimizer to train the network. The process of one training epoch is as follows: 1) randomly choose a sample from the training set, and convert the input and label into tensors; 2) feed the input to the network, and obtain the classification probability through the forward process; 3) calculate the network loss through `mse_loss`; 4) back-propagate the gradients, and update the training parameters; 5) judge whether the prediction is correct or not, and count the number of correct predictions to obtain the training accuracy every `plot_every` epochs; 6) evaluate the network on the testing set every `plot_every` epochs to obtain the testing accuracy. During training, we record the history of network loss `avg_losses` , training accuracy `accuracy_rec` and testing accuracy `test_accu_rec` , to observe the training process. After training, we will save the final state of the network for testing, and also some variables for later analyses.

```
# IF_TRAIN = 1
TRAIN_EPISODES = 1000000
plot_every = 1000
learning_rate = 1e-4

net = Net(n_letters, n_hidden, n_categories)
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)

print('Training...')
current_loss = 0
correct_num = 0
avg_losses = []
accuracy_rec = []
```

(下页继续)

(续上页)

```
test_accu_rec = []
start = time.time()
for epoch in range(1, TRAIN_EPISODES+1):
    net.train()
    category, line, category_tensor, line_tensor = randomPair('train')
    label_one_hot = F.one_hot(category_tensor.to(int), n_categories).float()

    optimizer.zero_grad()
    out_prob_log = net(line_tensor)
    loss = F.mse_loss(out_prob_log, label_one_hot)
    loss.backward()
    optimizer.step()

    current_loss += loss.data.item()

    guess, _ = categoryFromOutput(out_prob_log.data)
    if guess == category:
        correct_num += 1

    # Add current loss avg to list of losses
    if epoch % plot_every == 0:
        avg_losses.append(current_loss / plot_every)
        accuracy_rec.append(correct_num / plot_every)
        current_loss = 0
        correct_num = 0

    # evaluate the network on the testing set every ``plot_every`` epochs to obtain␣
↪the testing accuracy
    if epoch % plot_every == 0:  # int(TRAIN_EPISODES/1000)
        net.eval()
        with torch.no_grad():
            numCorrect = 0
            for i in range(n_categories):
                category = all_categories[i]
                for tname in category_lines_test[category]:
                    output = net(lineToTensor(tname))
                    guess, _ = categoryFromOutput(output.data)
                    if guess == category:
                        numCorrect += 1
            test_accu = numCorrect / testNumtot
            test_accu_rec.append(test_accu)
            print('Epoch %d %d%% (%s); Avg_loss %.4f; Train accuracy %.4f; Test␣
↪accuracy %.4f' % (
```

(下页继续)

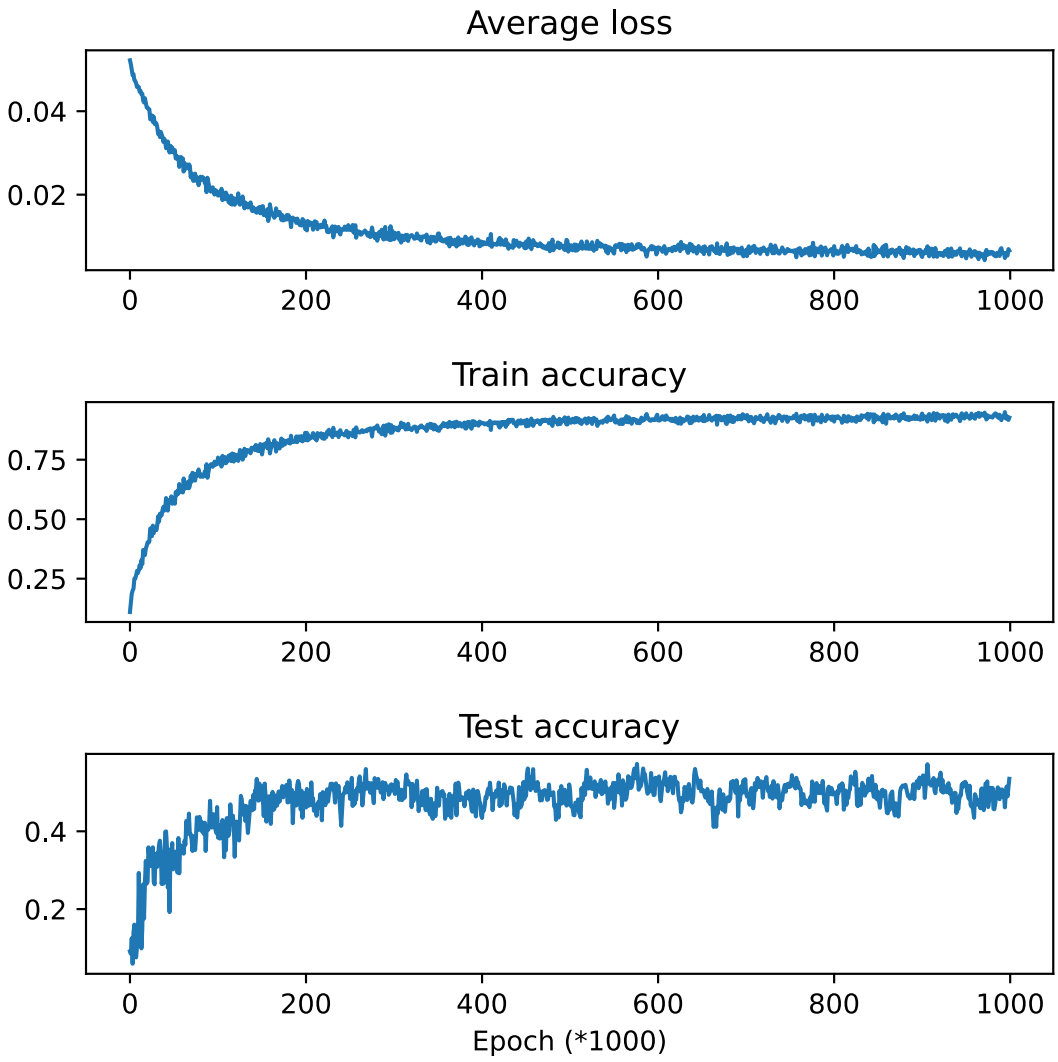
(续上页)

```
                epoch, epoch / TRAIN_EPISODES * 100, timeSince(start), avg_losses[-1],
↪ accuracy_rec[-1], test_accu))

torch.save(net, 'char_rnn_classification.pth')
np.save('avg_losses.npy', np.array(avg_losses))
np.save('accuracy_rec.npy', np.array(accuracy_rec))
np.save('test_accu_rec.npy', np.array(test_accu_rec))
np.save('category_lines_train.npy', category_lines_train, allow_pickle=True)
np.save('category_lines_test.npy', category_lines_test, allow_pickle=True)
# x = np.load('category_lines_test.npy', allow_pickle=True)  # way to loading the data
# xdict = x.item()

plt.figure()
plt.subplot(311)
plt.plot(avg_losses)
plt.title('Average loss')
plt.subplot(312)
plt.plot(accuracy_rec)
plt.title('Train accuracy')
plt.subplot(313)
plt.plot(test_accu_rec)
plt.title('Test accuracy')
plt.xlabel('Epoch (*1000)')
plt.subplots_adjust(hspace=0.6)
plt.savefig('TrainingProcess.svg')
plt.close()
```

We will observe the following results when executing `%run ./spiking_lstm_text.py` in Python Console with `IF_TRAIN = 1`.

```
Backend Qt5Agg is interactive backend. Turning interactive mode on.
Training...
Epoch 1000 0% (0m 18s); Avg_loss 0.0525; Train accuracy 0.0830; Test accuracy 0.0806
Epoch 2000 0% (0m 37s); Avg_loss 0.0514; Train accuracy 0.1470; Test accuracy 0.1930
Epoch 3000 0% (0m 55s); Avg_loss 0.0503; Train accuracy 0.1650; Test accuracy 0.0537
Epoch 4000 0% (1m 14s); Avg_loss 0.0494; Train accuracy 0.1920; Test accuracy 0.0938
...
...
Epoch 998000 99% (318m 54s); Avg_loss 0.0063; Train accuracy 0.9300; Test accuracy 0.
↪5036
Epoch 999000 99% (319m 14s); Avg_loss 0.0056; Train accuracy 0.9380; Test accuracy 0.
↪5004
Epoch 1000000 100% (319m 33s); Avg_loss 0.0055; Train accuracy 0.9340; Test accuracy␣
↪0.5118
```

The following picture shows how average loss `avg_losses` , training accuracy `accuracy_rec` and testing accuracy `test_accu_rec` improve with training.



Testing the network

We first load the well-trained network, and then conduct the following tests: 1) calculate the testing accuracy of the final network; 2) predict the language origin of the surnames provided by the user; 3) calculate the confusion matrix, indicating for every actual language (rows) which language the network guesses (columns).

```
# IF_TRAIN = 0
print('Testing...')
```

(下页继续)

(续上页)

```
net = torch.load('char_rnn_classification.pth')

# calculate the testing accuracy of the final network
print('Calculating testing accuracy...')
numCorrect = 0
for i in range(n_categories):
    category = all_categories[i]
    for tname in category_lines_test[category]:
        output = net(lineToTensor(tname))
        guess, _ = categoryFromOutput(output.data)
        if guess == category:
            numCorrect += 1
test_accu = numCorrect / testNumtot
print('Test accuracy: {:.3f}, Random guess: {:.3f}'.format(test_accu, 1/n_categories))

# predict the language origin of the surnames provided by the user
n_predictions = 3
for j in range(3):
    first_name = input('Please input a surname to predict its language origin:')
    print('\n> %s' % first_name)
    output = net(lineToTensor(first_name))

    # Get top N categories
    topv, topi = output.topk(n_predictions, 1, True)
    predictions = []

    for i in range(n_predictions):
        value = topv[0][i].item()
        category_index = topi[0][i].item()
        print('(%.2f) %s' % (value, all_categories[category_index]))
        predictions.append([value, all_categories[category_index]])

# calculate the confusion matrix
print('Calculating confusion matrix...')
confusion = torch.zeros(n_categories, n_categories)
n_confusion = 10000

# Keep track of correct guesses in a confusion matrix
for i in range(n_confusion):
    category, line, category_tensor, line_tensor = randomPair('all')
    output = net(line_tensor)
    guess, guess_i = categoryFromOutput(output.data)
    category_i = all_categories.index(category)
```

(下页继续)

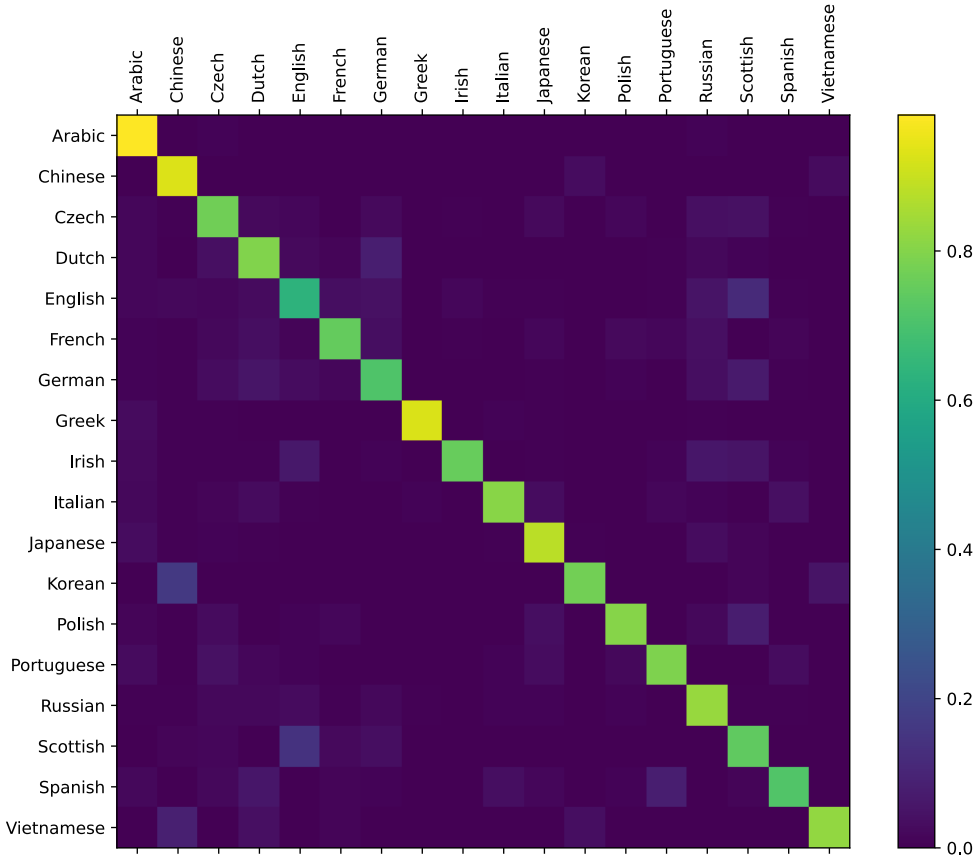
(续上页)

```
    confusion[category_i][guess_i] += 1

confusion = confusion / confusion.sum(1)
np.save('confusion.npy', confusion)

# Set up plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111)
cax = ax.matshow(confusion.numpy())
fig.colorbar(cax)
# Set up axes
ax.set_xticklabels(['] + all_categories, rotation=90)
ax.set_yticklabels(['] + all_categories)
# Force label at every tick
ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
ax.yaxis.set_major_locator(ticker.MultipleLocator(1))
# sphinx_gallery_thumbnail_number = 2
plt.show()
plt.savefig('ConfusionMatrix.svg')
plt.close()
```

We will observe the following results when executing `%run ./spiking_lstm_text.py` in Python Console with `IF_TRAIN = 0`.

```
Testing...
Calculating testing accuracy...
Test accuracy: 0.512, Random guess: 0.056
Please input a surname to predict its language origin:> YU
> YU
(0.18) Scottish
(0.12) English
(0.11) Italian
Please input a surname to predict its language origin:> Yu
> Yu
(0.63) Chinese
(0.23) Korean
(0.07) Vietnamese
Please input a surname to predict its language origin:> Zou
> Zou
(1.00) Chinese
(0.00) Arabic
(0.00) Polish
Calculating confusion matrix...
```

The following picture exhibits the confusion matrix, of which a brighter diagonal element indicates better prediction, and thus less confusion, such as Arabic and Greek. However, some languages are prone to confusion, such as Korean and Chinese, English and Scottish.



6.1.10 Propagation Pattern

Authors: fangwei123456

Single-Step and Multi-Step

Most modules in SpikingJelly (except for *spikingjelly.clock_driven.rnn*), e.g., *spikingjelly.clock_driven.layer.Dropout*, don't have a `MultiStep` prefix. These modules' `forward` functions define a single-step forward:

> Input X_t, output Y_t

If a module has a `MultiStep` prefix, e.g., *spikingjelly.clock_driven.layer.MultiStepDropout*, then this module's `forward` function defines the multi-step forward:

Input $X_t, t = 0, 1, ..., T-1$, output $Y_t, t = 0, 1, ..., T-1$

A single-step module can be easily packaged as a multi-step module. For example, we can use *spikingjelly.clock_driven.layer.MultiStepContainer*, which contains the origin module as a sub-module and implements the loop in time-steps in its `forward` function:

```
class MultiStepContainer(nn.Module):
    def __init__(self, module: nn.Module):
        super().__init__()
        self.module = module

    def forward(self, x_seq: torch.Tensor):
        y_seq = []
        for t in range(x_seq.shape[0]):
            y_seq.append(self.module(x_seq[t]))
            y_seq[-1].unsqueeze_(0)
        return torch.cat(y_seq, 0)

    def reset(self):
        if hasattr(self.module, 'reset'):
            self.module.reset()
```

Let us use *spikingjelly.clock_driven.layer.MultiStepContainer* to implement a multi-step IF neuron:

```
from spikingjelly.clock_driven import neuron, layer
import torch

neuron_num = 4
T = 8
if_node = neuron.IFNode()
x = torch.rand([T, neuron_num]) * 2
for t in range(T):
    print(f'if_node output spikes at t={t}', if_node(x[t]))
if_node.reset()

ms_if_node = layer.MultiStepContainer(if_node)
print("multi step if_node output spikes\n", ms_if_node(x))
ms_if_node.reset()
```

The outputs are:

```
if_node output spikes at t=0 tensor([1., 1., 1., 0.])
if_node output spikes at t=1 tensor([0., 0., 0., 1.])
if_node output spikes at t=2 tensor([1., 1., 1., 1.])
```

(下页继续)

(续上页)

```
if_node output spikes at t=3 tensor([0., 0., 1., 0.])
if_node output spikes at t=4 tensor([1., 1., 1., 1.])
if_node output spikes at t=5 tensor([1., 0., 0., 0.])
if_node output spikes at t=6 tensor([1., 0., 1., 1.])
if_node output spikes at t=7 tensor([1., 1., 1., 0.])
multi step if_node output spikes
 tensor([[1., 1., 1., 0.],
        [0., 0., 0., 1.],
        [1., 1., 1., 1.],
        [0., 0., 1., 0.],
        [1., 1., 1., 1.],
        [1., 0., 0., 0.],
        [1., 0., 1., 1.],
        [1., 1., 1., 0.]])
```

We can find that the single-step module and the multi-step module have the identical outputs.

Step-by-step and Layer-by-Layer

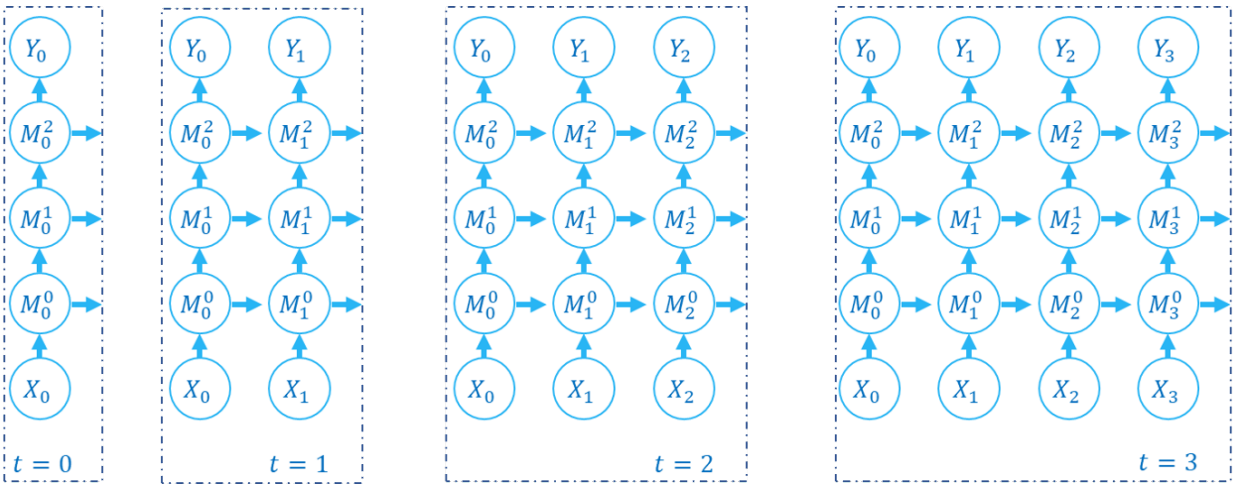
In the previous tutorials and examples, we run the SNNs *step-by-step*, e.g.,:

```
if_node = neuron.IFNode()
x = torch.rand([T, neuron_num]) * 2
for t in range(T):
    print(f'if_node output spikes at t={t}', if_node(x[t]))
```

step-by-step means that during the forward propagation, we firstly calculate the SNN' s outputs Y_0 at $t = 0$, then we calculate the SNN' s outputs Y_1 at $t = 1, \cdots$, and we can get the outputs at all time-steps $Y_t, t = 0, 1, ..., T - 1$. The followed code is a *step-by-step* example (we suppose `M0, M1, M2` are single-step modules):

```
net = nn.Sequential(M0, M1, M2)

for t in range(T):
    Y[t] = net(X[t])
```

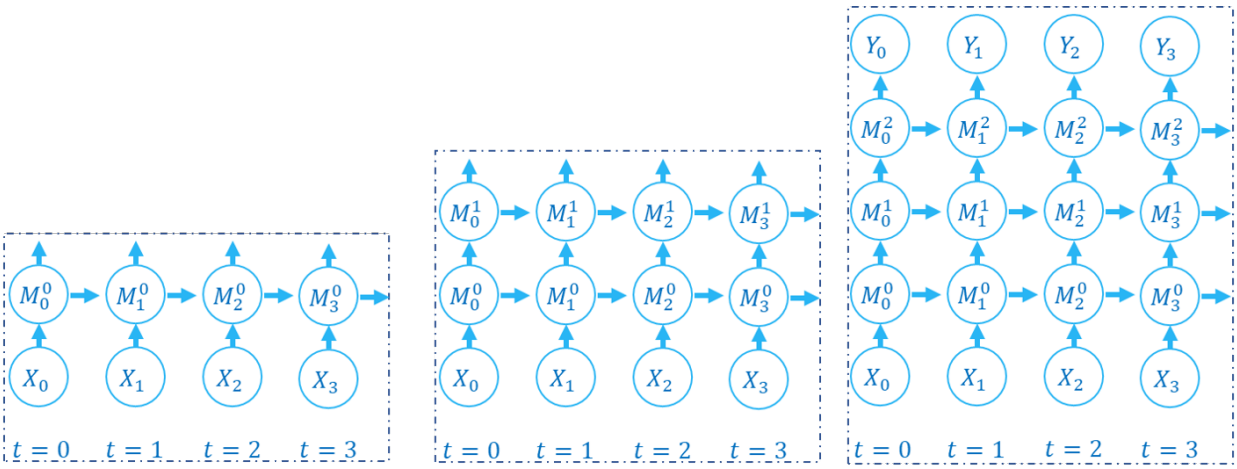
The computation graph of forward propagation is built as followed:



The forward propagation of SNN and RNN is along both spatial domain and temporal domain. *step-by-step* calculates states of the whole network step by step. We can also use an another order, which is *layer-by-layer*. *layer-by-layer* calculates states layer-by-layer. The followed code is a *layer-by-layer* example (we suppose `M0, M1, M2` are multi-step modules):

```
net = nn.Sequential(M0, M1, M2)

Y = net(X)
```

The computation graph of forward propagation is built as followed:



The *layer-by-layer* method is widely used in RNN and SNN, e.g., Low-activity supervised convolutional spiking neural networks applied to speech commands recognition calculates outputs of each layer to implement a temporal convolution. Their codes are availble at https://github.com/romainzimmer/s2net.

The difference between *step-by-step* and *layer-by-layer* is the order of traverse the computation graph. The computed results of both methods are exactly same. However, *step-by-step* has more degree of parallelism. When a layer is stateless, e.g., `torch.nn.Linear`, the *step-by-step* method may calculate as:

```
for t in range(T):
    y[t] = fc(x[t])  # x.shape=[T, batch_size, in_features]
```

The *layer-by-layer* method can calculate parallelly:

```
y = fc(x)  # x.shape=[T, batch_size, in_features]
```

For a stateless layer, we can concatenate inputs `shape=[T, batch_size, ...]` at time dimension as `shape=[T * batch_size, ...]` to avoid loop in time-steps. *spikingjelly.clock_driven.layer.SeqToANNContainer* has provided such a function in its `forward`. We can directly use this module:

```
with torch.no_grad():
    T = 16
    batch_size = 8
    x = torch.rand([T, batch_size, 4])
    fc = SeqToANNContainer(nn.Linear(4, 2), nn.Linear(2, 3))
    print(fc(x).shape)
```

The outputs are

```
torch.Size([16, 8, 3])
```

The outputs have `shape=[T, batch_size, ...]` and can be directly fed to the next layer.

6.1.11 Accelerate with CUDA-Enhanced Neuron and Layer-by-Layer Propagation

Authors: fangwei123456

CUDA-Enhanced Neuron

The neuron sharing the same name in `spikingjelly.cext.neuron` and `spikingjelly.clock_driven.neuron` do the same calculation when forward and backward. However, `spikingjelly.cext.neuron` fuses operations in one CUDA kernel, while `spikingjelly.clock_driven.neuron` uses PyTorch functions to perform operations and each PyTorch function needs to call a CUDA function, which causes grate overhead for calling CUDA kernels. Let us run a simple code to compare LIF neurons in both module:

```
from spikingjelly import cext
from spikingjelly.cext import neuron as cext_neuron
from spikingjelly.clock_driven import neuron, surrogate, layer
import torch


def cal_forward_t(multi_step_neuron, x, repeat_times):
```

(下页继续)

(续上页)

```
    with torch.no_grad():
        used_t = cext.cal_fun_t(repeat_times, x.device, multi_step_neuron, x)
        multi_step_neuron.reset()
        return used_t * 1000


def forward_backward(multi_step_neuron, x):
    multi_step_neuron(x).sum().backward()
    multi_step_neuron.reset()
    x.grad.zero_()


def cal_forward_backward_t(multi_step_neuron, x, repeat_times):
    x.requires_grad_(True)
    used_t = cext.cal_fun_t(repeat_times, x.device, forward_backward, multi_step_
↪neuron, x)
    return used_t * 1000


device = 'cuda:0'
lif = layer.MultiStepContainer(neuron.LIFNode(surrogate_function=surrogate.
↪ATan(alpha=2.0)))
lif_cuda = layer.MultiStepContainer(cext_neuron.LIFNode(surrogate_function='ATan',␣
↪alpha=2.0))
lif_cuda_tt = cext_neuron.MultiStepLIFNode(surrogate_function='ATan', alpha=2.0)
lif.to(device)
lif_cuda.to(device)
lif_cuda_tt.to(device)
N = 2 ** 20
print('forward')
lif.eval()
lif_cuda.eval()
lif_cuda_tt.eval()
for T in [8, 16, 32, 64, 128]:
    x = torch.rand(T, N, device=device)
    print(T, cal_forward_t(lif, x, 1024), cal_forward_t(lif_cuda, x, 1024), cal_
↪forward_t(lif_cuda_tt, x, 1024))

print('forward and backward')
lif.train()
lif_cuda.train()
lif_cuda_tt.train()
for T in [8, 16, 32, 64, 128]:
```

(下页继续)

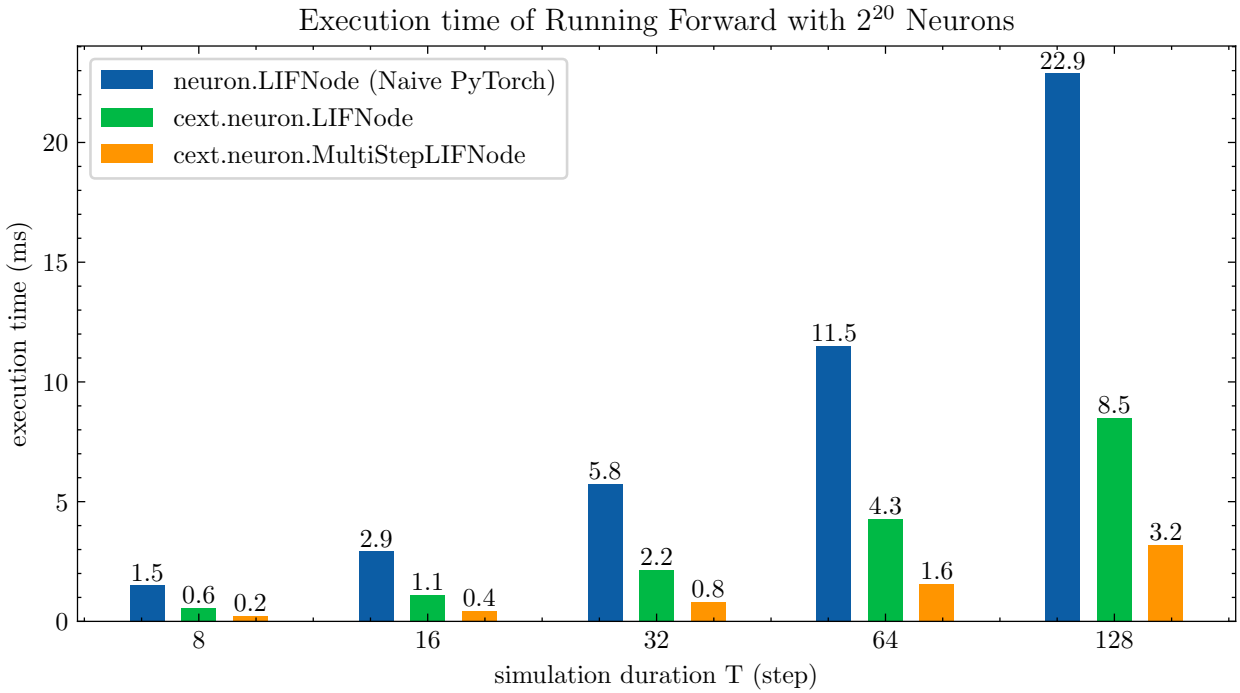
(续上页)

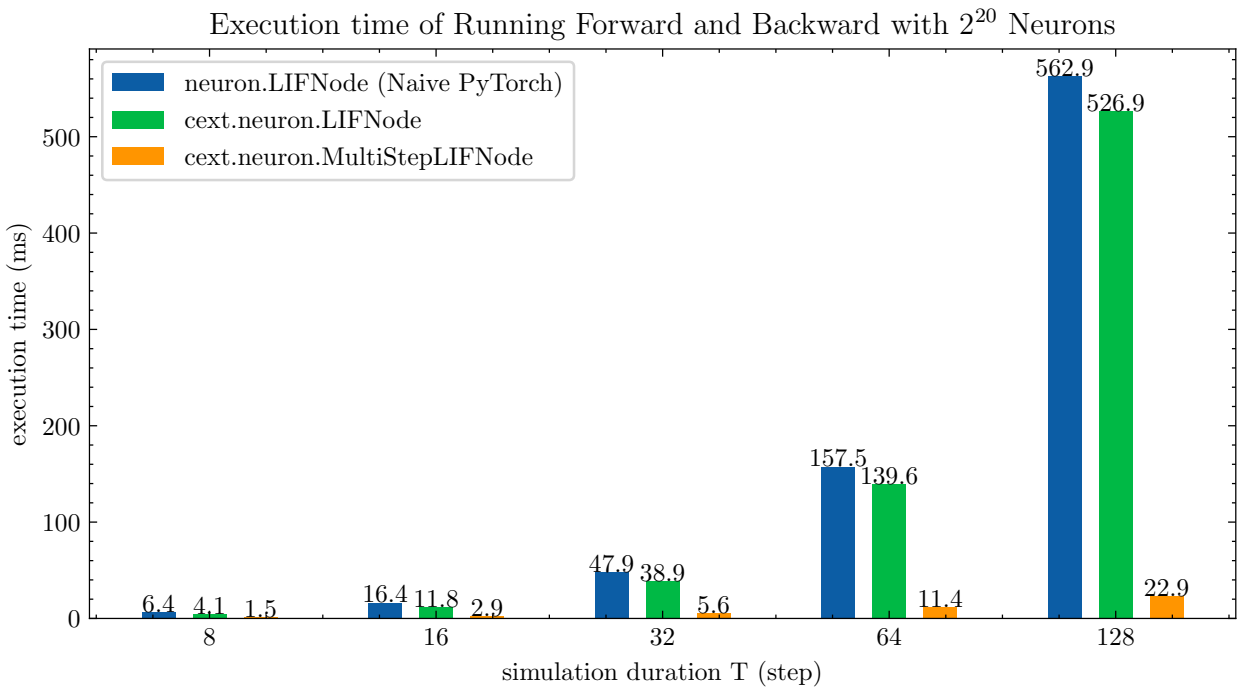
```
    x = torch.rand(T, N, device=device)
    print(T, cal_forward_backward_t(lif, x, 1024), cal_forward_backward_t(lif_cuda, x,
↪ 1024),
          cal_forward_backward_t(lif_cuda_tt, x, 1024))
```

The code is running at a Ubuntu server with *Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz* CPU and *GeForce RTX 2080 Ti* GPU. The outputs are:

```
forward
8 1.495931436011233 0.5668559867899603 0.23965466994013696
16 2.921243163427789 1.0935631946722424 0.42392046202621714
32 5.7503134660237265 2.1567279295595654 0.800143975766332
64 11.510705337741456 4.273202213653349 1.560730856454029
128 22.884282833274483 8.508097553431071 3.1778080651747587
forward and backward
8 6.444244811291355 4.052604411526772 1.4819166492543445
16 16.360167272978288 11.785220529191065 2.8625220465983148
32 47.86415797116206 38.88952818761027 5.645714411912195
64 157.53049964018828 139.59021832943108 11.367870506774125
128 562.9168437742464 526.8922436650882 22.945806705592986
```

We plot the results in a bar chart:





It can be found that neurons in `spikingjelly.cext.neuron` are faster than naive PyTorch neuron.

Accelerate Deep SNNs

Now let us use the CUDA-Enhanced Multi-Step neuron to re-implement the network in *Clock driven: Use convolutional SNN to identify Fashion-MNIST* and compare their speeds. There is no need to modify the training codes. We can only change the network' s codes:

```
class Net(nn.Module):
    def __init__(self, tau, T, v_threshold=1.0, v_reset=0.0):
        super().__init__()
        self.T = T

        self.static_conv = nn.Sequential(
            nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(128),
        )

        self.conv = nn.Sequential(
            cext_neuron.MultiStepIFNode(v_threshold=v_threshold, v_reset=v_reset,␣
↪surrogate_function='ATan', alpha=2.0),
            layer.SeqToANNContainer(
                    nn.MaxPool2d(2, 2),  # 14 * 14
                    nn.Conv2d(128, 128, kernel_size=3, padding=1, bias=False),
                    nn.BatchNorm2d(128),
```

(下页继续)

(续上页)

```
            ),
            cext_neuron.MultiStepIFNode(v_threshold=v_threshold, v_reset=v_reset,␣
↪surrogate_function='ATan', alpha=2.0),
        )
        self.fc = nn.Sequential(
            layer.SeqToANNContainer(
                    nn.MaxPool2d(2, 2),  # 7 * 7
                    nn.Flatten(),
            ),
            layer.MultiStepDropout(0.5),
            layer.SeqToANNContainer(nn.Linear(128 * 7 * 7, 128 * 3 * 3, bias=False)),
            cext_neuron.MultiStepLIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_
↪reset, surrogate_function='ATan', alpha=2.0),
            layer.MultiStepDropout(0.5),
            layer.SeqToANNContainer(nn.Linear(128 * 3 * 3, 128, bias=False)),
            cext_neuron.MultiStepLIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_
↪reset, surrogate_function='ATan', alpha=2.0),
            layer.SeqToANNContainer(nn.Linear(128, 10, bias=False)),
            cext_neuron.MultiStepLIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_
↪reset, surrogate_function='ATan', alpha=2.0)
        )


    def forward(self, x):
        x_seq = self.static_conv(x).unsqueeze(0).repeat(self.T, 1, 1, 1, 1)
        # [N, C, H, W] -> [1, N, C, H, W] -> [T, N, C, H, W]

        out_spikes_counter = self.fc(self.conv(x_seq)).sum(0)
        return out_spikes_counter / self.T
```

The fully codes are available at `spikingjelly.clock_driven.examples.conv_fashion_mnist_cuda_lbl`. Run this example with the same arguments and devices as those in *Clock driven: Use convolutional SNN to identify Fashion-MNIST*. The outputs are:

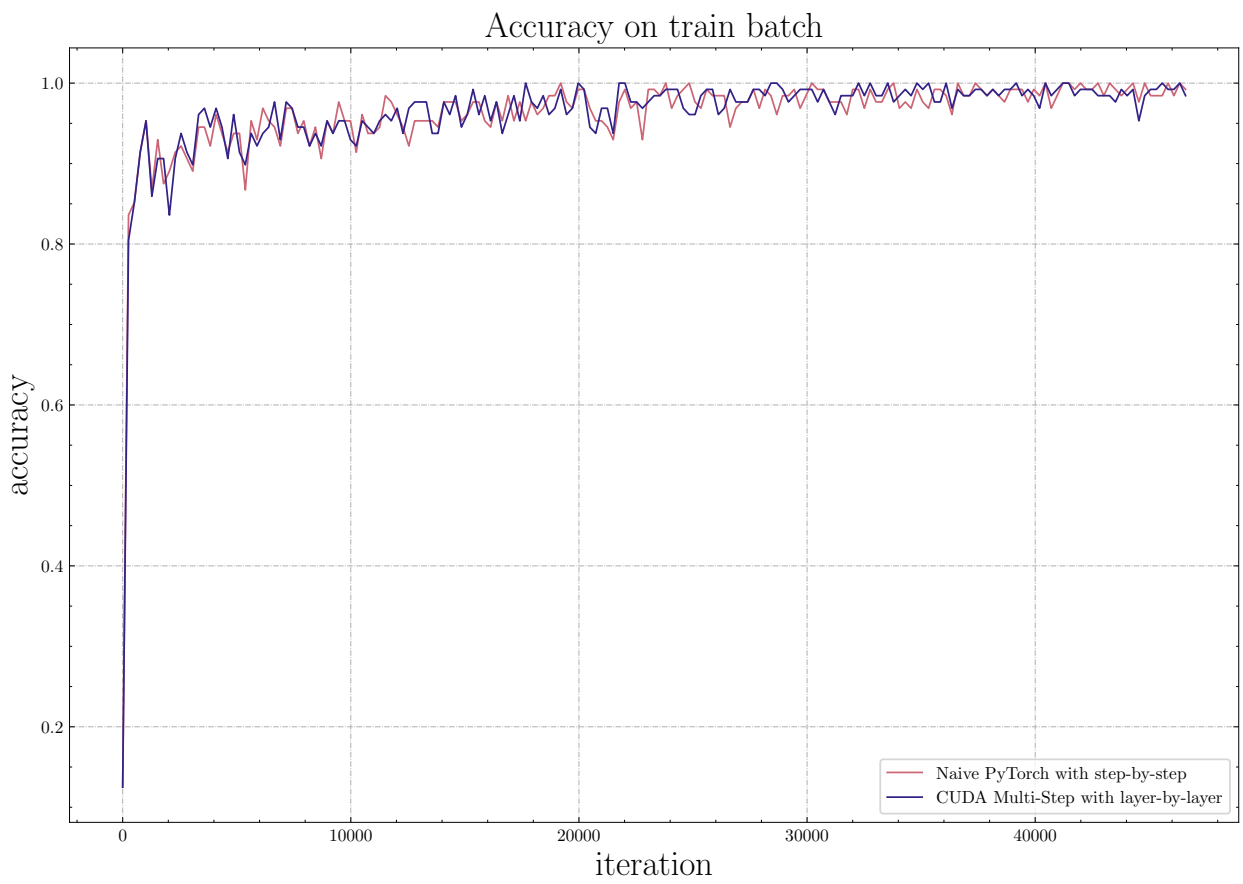
```
saving net...
saved
epoch=0, t_train=26.745780434459448, t_test=1.4819979975000024, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2,␣
↪max_test_accuracy=0.8705, train_times=468
saving net...
saved
epoch=1, t_train=26.087690989486873, t_test=1.502928489819169, device=cuda:0, dataset_
↪dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2, max_test_
↪accuracy=0.8913, train_times=936
```

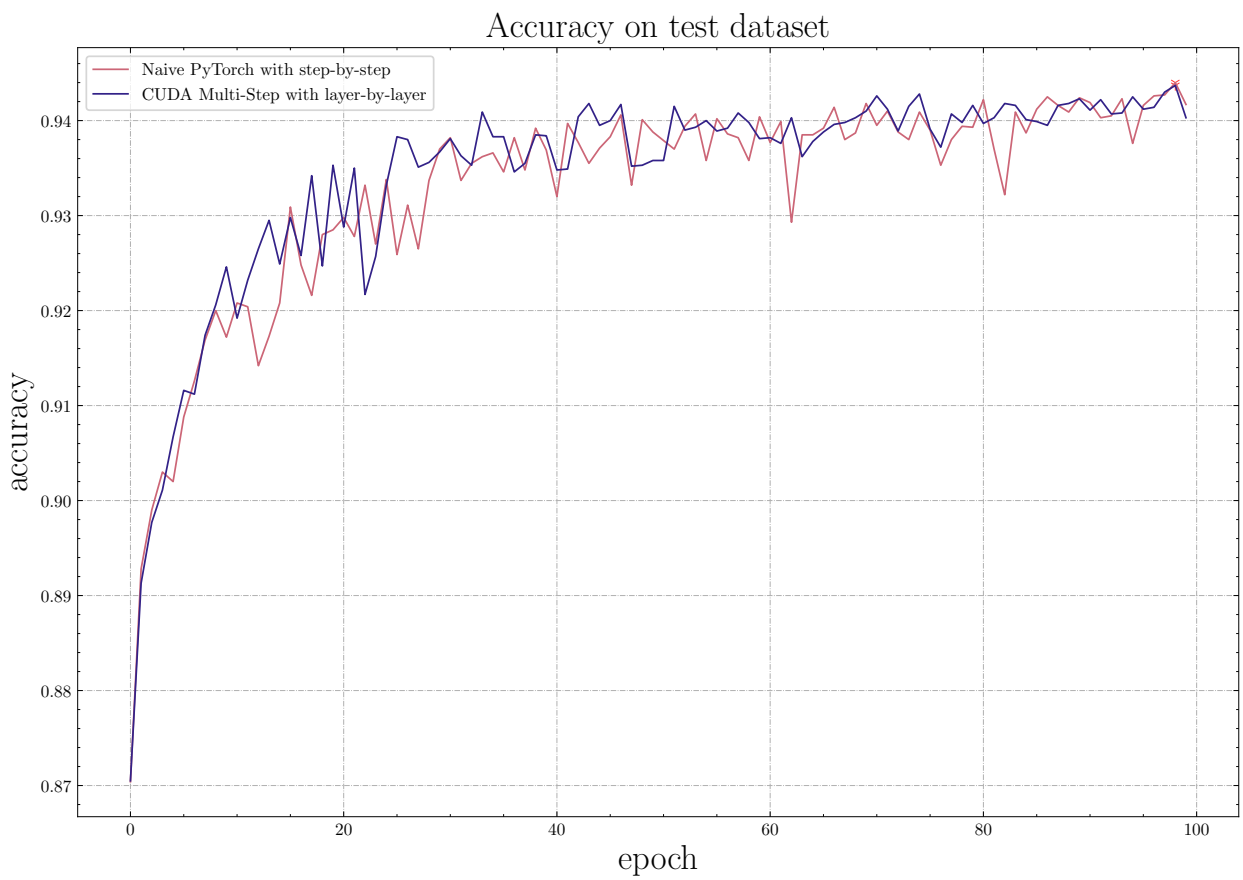
(下页继续)

(续上页)

```
saving net...
saved
epoch=2, t_train=26.281963238492608, t_test=1.4901704853400588, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2,␣
↪max_test_accuracy=0.8977, train_times=1404
saving net...
saved

...

epoch=96, t_train=26.286096683703363, t_test=1.5033660298213363, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2,␣
↪max_test_accuracy=0.9428, train_times=45396
saving net...
saved
epoch=97, t_train=26.185854725539684, t_test=1.4934641849249601, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2,␣
↪max_test_accuracy=0.943, train_times=45864
saving net...
saved
epoch=98, t_train=26.256993867456913, t_test=1.5093903196975589, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2,␣
↪max_test_accuracy=0.9437, train_times=46332
epoch=99, t_train=26.200945735909045, t_test=1.4959839908406138, device=cuda:0,␣
↪dataset_dir=./fmnist, batch_size=128, learning_rate=0.001, T=8, log_dir=./logs2,␣
↪max_test_accuracy=0.9437, train_times=46800
```

The highest accuracy on test dataset is 94.37%, which is very close to 94.4% in *Accelerate with CUDA-Enhanced Neuron and Layer-by-Layer Propagation*. The accuracy curves on training batch and test dataset during training are as followed:


Accuracy on train batch
accuracy
iteration
1.0
0.8
0.6
0.4
0.2
0
10000
20000
30000
40000
Naive PyTorch with step-by-step
CUDA Multi-Step with layer-by-layer



In fact, we set an identical seed in both examples, but get a different results, which maybe caused by the numerical errors between CUDA and PyTorch functions. The logs also record the execution time of training and testing. It can be found that the training execution time of the SNN with CUDA-Enhanced neurons and Layer-by-Layer propagation is 64% of the naive PyTorch SNN, and the testing execution time is 58% of the naive PyTorch SNN.

6.1.12 Neuromorphic Datasets Processing

Authors: fangwei123456

`spikingjelly.datasets` provides frequently-used neuromorphic datasets, including N-MNIST[1], CIFAR10-DVS[2], DVS128 Gesture[3], NAV Gesture[4], ASLDVS[5], etc. In this tutorial, we will take DVS 128 Gesture dataset as an example to show how to use SpikingJelly to process neuromorphic datasets.

[1] Orchard, Garrick, et al. "Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades." Frontiers in Neuroscience, vol. 9, 2015, pp. 437–437.

[2] Li, Hongmin, et al. "CIFAR10-DVS: An Event-Stream Dataset for Object Classification." Frontiers in Neuroscience, vol. 11, 2017, pp. 309–309.

[3] Amir, Arnon, et al. "A Low Power, Fully Event-Based Gesture Recognition System." 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 7388–7397.

[4] Maro, Jean-Matthieu, et al. "Event-Based Visual Gesture Recognition with Background Suppression Running on a Smart-Phone." 2019 14th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2019), 2019, p. 1.

[5] Bi, Yin, et al. "Graph-Based Object Classification for Neuromorphic Vision Sensing." 2019 IEEE/CVF International Conference on Computer Vision (ICCV), 2019, pp. 491–501.

Download DVS128 Gesture

The DVS128 Gesture dataset can be downloaded from https://ibm.ent.box.com/s/3hiq58ww1pbbjrinh367ykfdf60xsfm8/folder/50167556794. The box website does not allow us to download data by python codes without login. Thus, the user have to download manually. Suppose we have downloaded the dataset into `E:/datasets/DVS128Gesture`, then the directory structure is

```
.
|-- DvsGesture.tar.gz
|-- LICENSE.txt
|-- README.txt
`-- gesture_mapping.csv
```

Get Events Data

Let us import DVS128 Gesture from SpikingJelly and create train/test set. We set `use_frame=False` to use Event data rather than frame data.

```
from spikingjelly.datasets import DVS128Gesture

root_dir = 'E:/datasets/DVS128Gesture'
train_set = DVS128Gesture(root_dir, train=True, use_frame=False)
test_set = DVS128Gesture(root_dir, train=True, use_frame=False)
```

SpikingJelly will do the followed work when running these codes:

1. Check whether the dataset exists. If the dataset exists, check MD5 to ensure the dataset is complete. Then SpikingJelly will extract the origin data into the `extracted` folder
2. The sample in DVS128 Gesture is the video which records one actor displayed different gestures under different illumination conditions. Hence, an AER sample contains many gestures and there is also a adjoint csv file to label the time stamp of each gesture. Hence, an AER sample is not a sample with one class but multi-classes. SpikingJelly will use multi-threads to cut and extract each gesture from these files.

Here are the terminal outputs:

```
DvsGesture.tar.gz already exists, check md5
C:/Users/fw/anaconda3/envs/pytorch-env/lib/site-packages/torchaudio/backend/utils.
↪py:88: UserWarning: No audio backend is available.
  warnings.warn('No audio backend is available.')
md5 checked, extracting...
mkdir E:/datasets/DVS128Gesture/events_npy
mkdir E:/datasets/DVS128Gesture/events_npy/train
mkdir E:/datasets/DVS128Gesture/events_npy/test
read events data from *.aedat and save to *.npy...
```

(下页继续)

(续上页)

```
convert events data from aedat to numpy format.
thread 0 start
thread 1 start
thread 2 start
thread 3 start
thread 4 start
thread 5 start
thread 6 start
thread 7 start
  0%|          | 0/122 [00:00<?, ?it/s]working thread: [0, 1, 2, 3, 4, 5, 6, 7]
finished thread: []
```

We have to wait for a moment because the cutting and extracting is very slow. A `events_npy` folder will be created and contain the train/test set:

```
|-- events_npy
|   |-- test
|   `-- train
```

Print a sample:

```
x, y = train_set[0]
print('event', x)
print('label', y)
```

The output is:

```
event {'t': array([172843814, 172843824, 172843829, ..., 179442748, 179442763,
       179442789]), 'x': array([ 54,  59,  53, ...,  36, 118, 118]), 'y': array([116,␣
↪113,  92, ..., 102,  80,  83]), 'p': array([0, 1, 1, ..., 0, 1, 1])}
label 9
```

where `x` is a dictionary with keys `['t', 'x', 'y', 'p']`;"y" is the label of the sample. Note that the classes number of DVS128 Gesture is 11.

Get Frames Data

The event-to-frame integrating method for pre-processing neuromorphic datasets is widely used. We use the same method from[6] in SpikingJelly. Data in neuromorphic datasets are in the formulation of $E(x_i, y_i, t_i, p_i)$ that represent the event' s coordinate, time and polarity. We split the event' s number N into T slices with nearly the same number of events in each slice and integrate events to frames. Note that T is also the simulating time-step. Denote a two channels frame as

[6] Fang, Wei, et al. "Incorporating Learnable Membrane Time Constant to Enhance Learning of Spiking Neural Networks." ArXiv: Neural and Evolutionary Computing, 2020.

$F(j)$ and a pixel at (p, x, y) as $F(j, p, x, y)$, the pixel value is integrated from the events data whose indices are between j_l and j_r:

$$
\begin{aligned}
j_l &= \left\lfloor \frac{N}{T} \right\rfloor \cdot j \\
j_r &= \begin{cases} \left\lfloor \frac{N}{T} \right\rfloor \cdot (j+1), & \text{if } j < T-1 \\ N, & \text{if } j = T-1 \end{cases} \\
F(j, p, x, y) &= \sum_{i=j_l}^{j_r-1} \mathcal{I}_{p,x,y}(p_i, x_i, y_i)
\end{aligned}
$$

where $\lfloor \cdot \rfloor$ is the floor operation, $\mathcal{I}_{p,x,y}(p_i, x_i, y_i)$ is an indicator function and it equals 1 only when $(p, x, y) = (p_i, x_i, y_i)$.

SpikingJelly will integrate events to frames when running the followed codes:

```
train_set = DVS128Gesture(root_dir, train=True, use_frame=True, frames_num=20, split_
↪by='number', normalization=None)
test_set = DVS128Gesture(root_dir, train=True, use_frame=True, frames_num=20, split_
↪by='number', normalization=None)
```

The outputs from the terminal are:

```
npy format events data root E:/datasets/DVS128Gesture/events_npy/train, E:/datasets/
↪DVS128Gesture/events_npy/test already exists
mkdir E:/datasets/DVS128Gesture/frames_num_20_split_by_number_normalization_None, E:/
↪datasets/DVS128Gesture/frames_num_20_split_by_number_normalization_None/train, E:/
↪datasets/DVS128Gesture/frames_num_20_split_by_number_normalization_None/test.
creating frames data..
thread 0 start, processing files index: 0 : 294.
thread 1 start, processing files index: 294 : 588.
thread 2 start, processing files index: 588 : 882.
thread 4 start, processing files index: 882 : 1176.
thread 0 finished.
thread 1 finished.
thread 2 finished.
thread 3 finished.
thread 0 start, processing files index: 0 : 72.
thread 1 start, processing files index: 72 : 144.
thread 2 start, processing files index: 144 : 216.
thread 4 start, processing files index: 216 : 288.
thread 0 finished.
thread 1 finished.
thread 2 finished.
thread 3 finished.
```

A `frames_num_20_split_by_number_normalization_None` folder will be created and contain the Frame data.

Print a sample:

```
x, y = train_set[0]
x, y = train_set[0]
print('frame shape', x.shape)
print('label', y)
```

The output is:

```
frame shape torch.Size([20, 2, 128, 128])
label 9
```

Let us visualize a sample:

```
from torchvision import transforms
from matplotlib import pyplot as plt

x, y = train_set[5]
to_img = transforms.ToPILImage()

img_tensor = torch.zeros([x.shape[0], 3, x.shape[2], x.shape[3]])
img_tensor[:, 1] = x[:, 0]
img_tensor[:, 2] = x[:, 1]


for t in range(img_tensor.shape[0]):
    print(t)
    plt.imshow(to_img(img_tensor[t]))
    plt.pause(0.01)
```

We will get the images like:

6.2 Modules Docs

- *APIs*

6.3 Indices and tables

- Index
- Module Index
- Search Page

6.4 Citation

If you use SpikingJelly in your work, please cite it as follows:

```
@misc{SpikingJelly,
    title = {SpikingJelly},
    author = {Fang, Wei and Chen, Yanqi and Ding, Jianhao and Chen, Ding and Yu,␣
↪Zhaofei and Zhou, Huihui and Tian, Yonghong and other contributors},
    year = {2020},
    publisher = {GitHub},
    journal = {GitHub repository},
    howpublished = {\url{https://github.com/fangwei123456/spikingjelly}},
}
```

6.5 About

Multimedia Learning Group, Institute of Digital Media (NELVT), Peking University and Peng Cheng Laboratory are the main developers of SpikingJelly.

The list of developers can be found at contributors.

6.5.1 spikingjelly.clock_driven package

spikingjelly.clock_driven.examples package

Submodules

spikingjelly.clock_driven.examples.lif_fc_mnist module

spikingjelly.clock_driven.examples.lif_fc_mnist.**main**()

- *API in English*

返回 None

使用全连接-LIF-全连接-LIF 的网络结构，进行 MNIST 识别。这个函数会初始化网络进行训练，并显示训练过程中在测试集的正确率。

- *中文 API*

The network with FC-LIF-FC-LIF structure for classifying MNIST. This function initials the network, starts training and shows accuracy on test dataset.

spikingjelly.clock_driven.examples.conv_fashion_mnist module

class spikingjelly.clock_driven.examples.conv_fashion_mnist.**Net**(*tau*, *T*, *v_threshold=1.0*, *v_reset=0.0*)

基类：`torch.nn.modules.module.Module`

forward(*x*)

training: bool

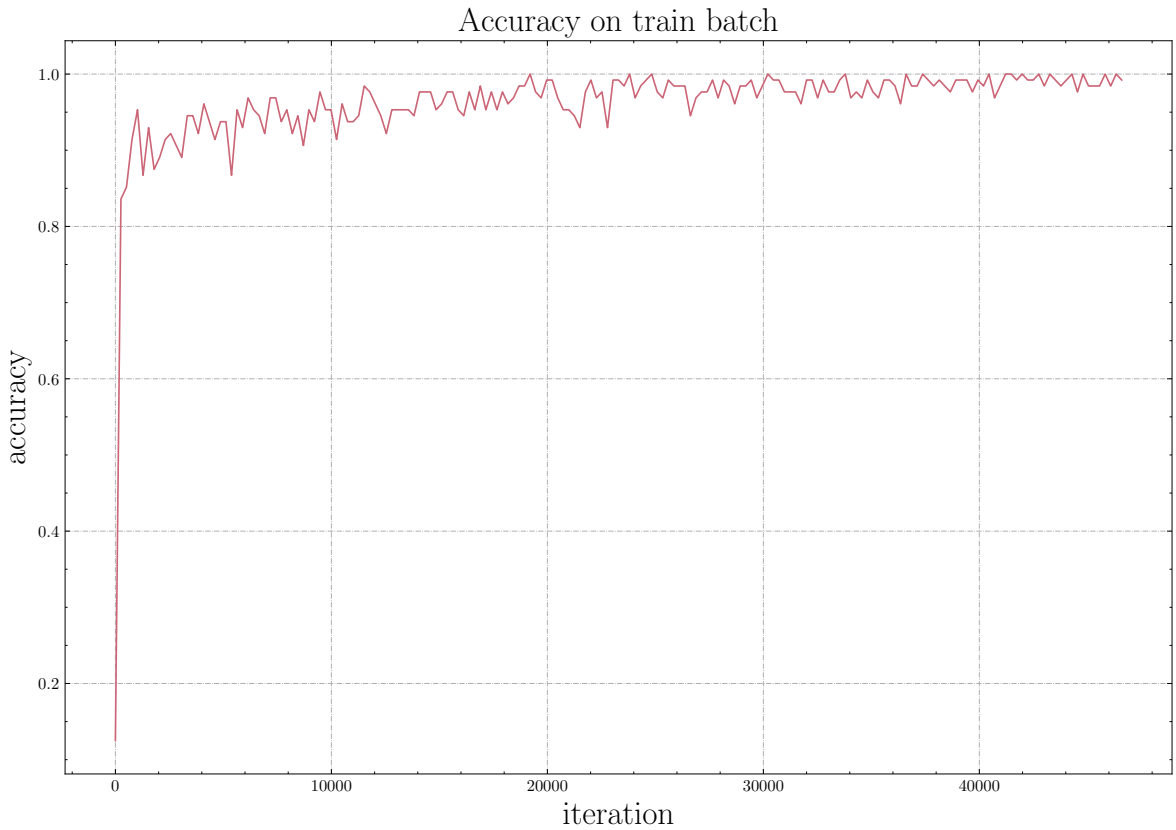
spikingjelly.clock_driven.examples.conv_fashion_mnist.**main**()

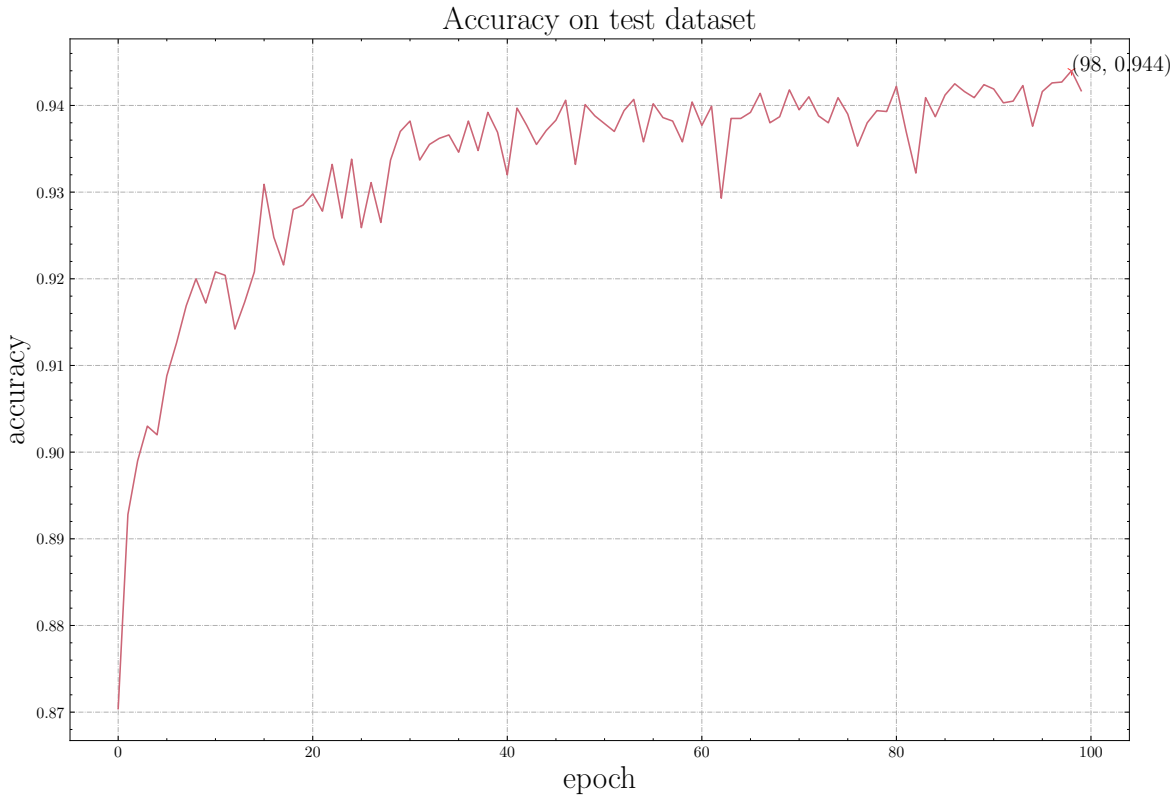
- *API in English*

返回 None

使用卷积-全连接的网络结构，进行 Fashion MNIST 识别。这个函数会初始化网络进行训练，并显示训练过程中在测试集的正确率。会将训练过程中测试集正确率最高的网络保存在 `tensorboard` 日志文件的同级目录下。这个目录的位置，是在运行 `main()` 函数时由用户输入的。

训练 100 个 epoch，训练 batch 和测试集上的正确率如下：

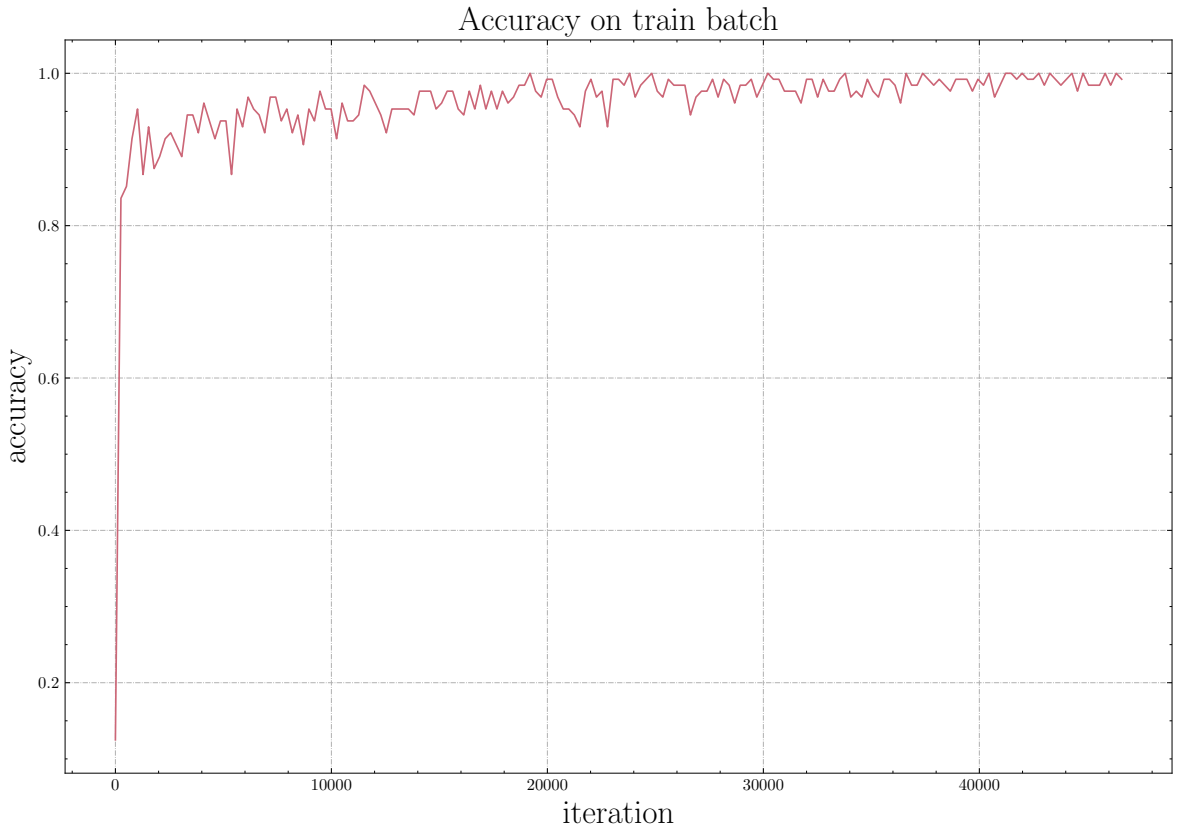

Accuracy on train batch
accuracy
iteration
1.0
0.8
0.6
0.4
0.2
0
10000
20000
30000
40000

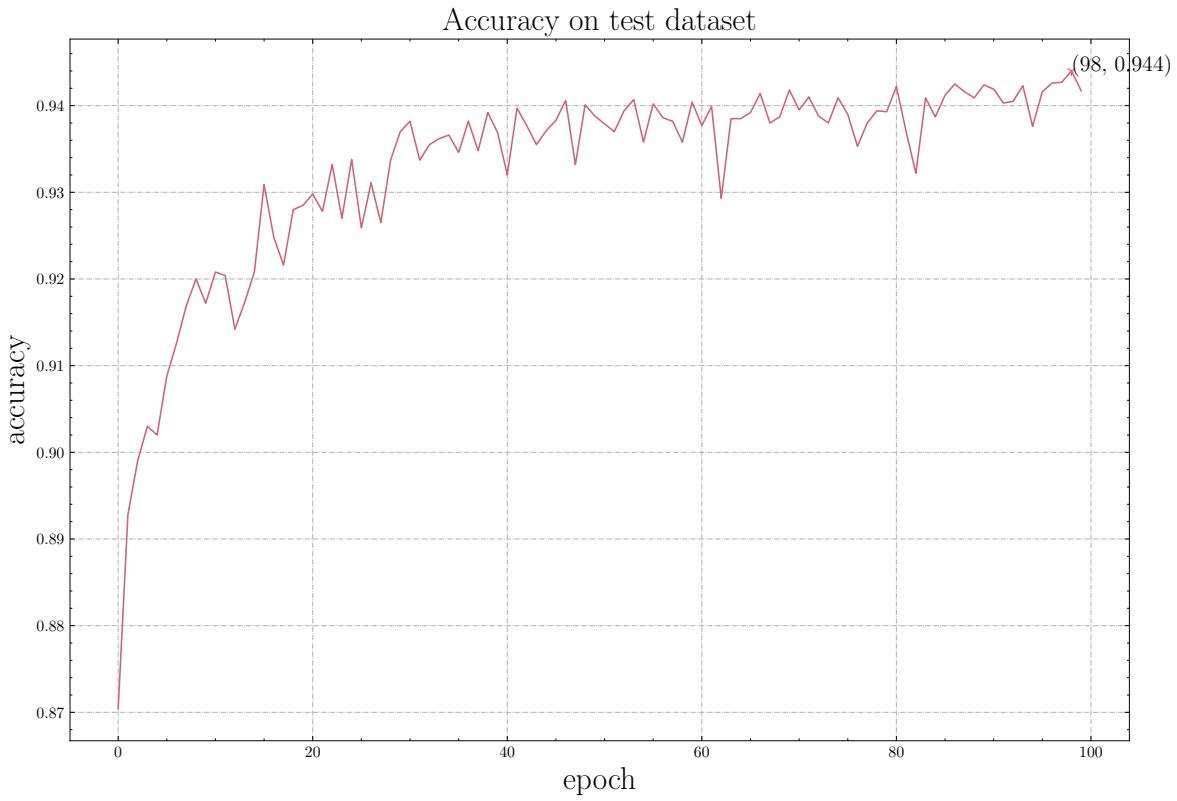

Accuracy on test dataset
(98, 0.944)
accuracy
epoch
0.94
0.93
0.92
0.91
0.90
0.89
0.88
0.87
0
20
40
60
80
100

- *中文 API*

The network with Conv-FC structure for classifying Fashion MNIST. This function initials the network, starts training and shows accuracy on test dataset. The net with the max accuracy on test dataset will be saved in the root directory for saving `tensorboard` logs, which is inputted by user when running the `main()` function.

After 100 epochs, the accuracy on train batch and test dataset is as followed:





spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation module

代码作者： *Yanqi Chen* <*chyq@pku.edu.cn*>

A reproduction of the paper Enabling Spike-Based Backpropagation for Training Deep Neural Network Architectures.

This code reproduces a novel gradient-based training method of SNN. We to some extent refer to the network structure and some other detailed implementation in the authors' implementation. Since the training method and neuron models are slightly different from which in this framework, we rewrite them in a compatible style.

Assuming you have at least 1 Nvidia GPU.

class `spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation.re`

基类：`torch.autograd.function.Function`

static forward(*ctx*, *x*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation.**Ba**

基类: torch.nn.modules.module.Module

spiking()

forward(*dv: torch.Tensor*)

reset()

training: bool

class spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation.**L**

基类：*spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation BaseNode*

forward(*dv: torch.Tensor*)

training: bool

class spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation.**IF**

基类：*spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation BaseNode*

forward(*dv: torch.Tensor*)

training: bool

class spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation.**Re**

基类：torch.nn.modules.module.Module

forward(*x*)

reset_()

training: bool

spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation.**main**()

Module contents

spikingjelly.clock_driven.encoding package

Module contents

class spikingjelly.clock_driven.encoding.**BaseEncoder**

基类：torch.nn.modules.module.Module

所有编码器的基类。编码器将输入数据（例如图像）编码为脉冲数据。

forward(*x*)

参数 **x** –要编码的数据

返回 编码后的脉冲，或者是 None

将 x 编码为脉冲。少数编码器（例如 ConstantEncoder）可以将 x 编码成时长为 1 个 dt 的脉冲，在这种情况下，本函数返回编码后的脉冲。

多数编码器（例如 PeriodicEncoder），都是把 x 编码成时长为 n 个 dt 的脉冲 out_spike，out_spike.shape=[n, *]。

因此编码一次后，需要调用 n 次 step() 函数才能将脉冲全部发放完毕，第 index 次调用 step() 会得到 out_spike[index]。

step()

返回 1 个 dt 的脉冲

多数编码器（例如 PeriodicEncoder），编码一次 x，需要经过多步仿真才能将数据输出，这种情况下则用 step 来获取每一步的数据。

reset()

返回 None

将编码器的所有状态变量设置为初始状态。对于有状态的编码器，需要重写这个函数。

training: **bool**

class spikingjelly.clock_driven.encoding.**PeriodicEncoder**(*out_spike*)

基类：*spikingjelly.clock_driven.encoding.BaseEncoder*

参数 **out_spike** –shape=[T, *]，PeriodicEncoder 会不断的输出 out_spike[0], out_spike[1], ⋯, out_spike[T-1], out_spike[0], out_spike[1], ⋯

给定 out_spike 后，周期性的输出 out_spike[0], out_spike[1], ⋯, out_spike[T-1] 的编码器。

forward(*x*)

参数 **x** –输入数据，实际上并不需要输入数据，因为 out_spike 在初始化时已经被指定了

返回 调用 step() 后得到的返回值

step()

返回 out_spike[index]

初始化时 index=0，每调用一次，index 则自增 1，index 为 T 时修改为 0。

set_out_spike(*out_spike*)

参数 **out_spike** –新设定的 out_spike，必须是 torch.bool

返回 None

重新设定编码器的输出脉冲 self.out_spike 为 out_spike。

reset()

返回 None

重置编码器的状态变量，对于 PeriodicEncoder 而言将索引 index 置 0 即可。

training: **bool**

class spikingjelly.clock_driven.encoding.**LatencyEncoder**(*max_spike_time*, *function_type='linear'*, *device='cpu'*)

基类：*spikingjelly.clock_driven.encoding.BaseEncoder*

参数

- **max_spike_time** –最晚（最大）脉冲发放时间
- **function_type** –'linear' 或 'log'
- **device** –数据所在设备

延迟编码，刺激强度越大，脉冲发放越早。要求刺激强度已经被归一化到 [0, 1]。

脉冲发放时间 t_i 与刺激强度 x_i 满足：

type='linear'

$$t_i = (t_{max} - 1) * (1 - x_i)$$

type='log'

$$t_i = (t_{max} - 1) - ln(\alpha * x_i + 1)$$

α 满足：

$$(t_{max} - 1) - ln(\alpha * 1 + 1) = 0$$

这导致此编码器很容易发生溢出，因为

$$\alpha = e^{t_{max}-1} - 1$$

当 t_{max} 较大时 α 极大。

示例代码：

```
x = torch.rand(size=[3, 2])
max_spike_time = 20
le = encoding.LatencyEncoder(max_spike_time)

le(x)
```

(下页继续)

(续上页)

```
print(x)
print(le.spike_time)
for i in range(max_spike_time):
    print(le.step())
```

forward(*x*)

参数 **x** –要编码的数据，任意形状的 tensor，要求 x 的数据范围必须在 [0, 1]

将输入数据 x 编码为 max_spike_time 个时刻的 max_spike_time 个脉冲。

step()

返回 out_spike[index]

初始化时 index=0，每调用一次，index 则自增 1，index 为 max_spike_time 时修改为 0。

reset()

返回 None

重置 LatencyEncoder 的所有状态变量（包括 spike_time，out_spike，index）为初始值 0。

training: bool

class spikingjelly.clock_driven.encoding.**PoissonEncoder**

基类：*spikingjelly.clock_driven.encoding.BaseEncoder*

泊松频率编码，输出脉冲可以看作是泊松流，发放脉冲的概率即为刺激强度，要求刺激强度已经被归一化到 [0, 1]。

示例代码：

```
pe = encoding.PoissonEncoder()
x = torch.rand(size=[8])
print(x)
for i in range(10):
    print(pe(x))
```

forward(*x*)

参数 **x** –要编码的数据，任意形状的 tensor，要求 x 的数据范围必须在 [0, 1]

将输入数据 x 编码为脉冲，脉冲发放的概率即为对应位置元素的值。

training: bool

class spikingjelly.clock_driven.encoding.**GaussianTuningCurveEncoder**(*x_min*, *x_max*, *tuning_curve_num*, *max_spike_time*, *device='cpu'*)

基类：*spikingjelly.clock_driven.encoding.BaseEncoder*

参数

- **x_min** –float，或者是 shape=[M] 的 tensor，表示 M 个特征的最小值
- **x_max** –float，或者是 shape=[M] 的 tensor，表示 M 个特征的最大值
- **tuning_curve_num** –编码每个特征使用的高斯函数（调谐曲线）数量
- **max_spike_time** –最大脉冲发放时间，所有数据都会被编码到 [0, max_spike_time - 1] 范围内的脉冲发放时间
- **device** –数据所在设备

Bohte S M, Kok J N, La Poutre H. Error-backpropagation in temporally encoded networks of spiking neurons[J]. Neurocomputing, 2002, 48(1-4): 17-37.

高斯调谐曲线编码，一种时域编码方法。

首先生成 tuning_curve_num 个高斯函数，这些高斯函数的对称轴在数据范围内均匀排列，对于每一个输入 x，计算 tuning_curve_num 个高斯函数的值，使用这些函数值线性地生成 tuning_curve_num 个脉冲发放时间。

待编码向量是 M 维 tensor，也就是有 M 个特征。

1 个 M 维 tensor 会被编码成 shape=[M, tuning_curve_num] 的 tensor，表示 M * tuning_curve_num 个神经元的脉冲发放时间。

需要注意的是，编码一次数据，经过 max_spike_time 步仿真，才能进行下一次的编码。

示例代码：

```
x = torch.rand(size=[3, 2])
tuning_curve_num = 10
max_spike_time = 20
ge = encoding.GaussianTuningCurveEncoder(x.min(0)[0], x.max(0)[0], tuning_curve_
↪num=tuning_curve_num, max_spike_time=max_spike_time)
ge(x)
for i in range(max_spike_time):
    print(ge.step())
```

forward(*x*)

参数 **x** –要编码的数据，shape=[batch_size, M]

将输入数据 x 编码为脉冲。

step()

返回 out_spike[index]

初始化时 index=0，每调用一次，index 则自增 1，index 为 max_spike_time 时修改为 0。

reset()

返回 None

重置 GaussianTuningCurveEncoder 的所有状态变量（包括 spike_time，out_spike，index）为初始值 0。

training: bool

class spikingjelly.clock_driven.encoding.**IntervalEncoder**(*T_in*, *shape*, *device='cpu'*)

基类：*spikingjelly.clock_driven.encoding.BaseEncoder*

参数

- **T_in** –脉冲发放的间隔
- **shape** –输出形状
- **device** –输出脉冲所在的设备

每隔 T_in 个步长就发放一次脉冲的编码器。

step()

reset()

training: bool

class spikingjelly.clock_driven.encoding.**WeightedPhaseEncoder**(*phase*, *period*, *device='cpu'*)

基类：*spikingjelly.clock_driven.encoding.BaseEncoder*

参数

- **phase** (*int*) –一个周期内用于编码的相数
- **device** (*str*) –输出脉冲所在的设备

Kim J, Kim H, Huh S, et al. Deep neural networks with weighted spikes[J]. Neurocomputing, 2018, 311: 373-386.

带权的相位编码，一种基于二进制表示的编码方法。

将输入按照二进制各位展开，从高位到低位遍历输入进行脉冲编码。相比于频率编码，每一位携带的信息量更多。编码相位数为 K 时，可以对于处于区间 $[0, 1-2^{-K}]$ 的数进行编码。以下为原始论文中的示例：

Phase (K=8)	1	2	3	4	5	6	7	8
Spike weight $\omega(t)$	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
192/256	1	1	0	0	0	0	0	0
1/256	0	0	0	0	0	0	0	1
128/256	1	0	0	0	0	0	0	0
255/256	1	1	1	1	1	1	1	1

forward(*x*)

参数 **x** –要编码的数据，shape=[batch_size, *]

将输入数据 x 编码为一个周期内的脉冲。

step()

reset()

training: bool

spikingjelly.clock_driven.functional package

Module contents

spikingjelly.clock_driven.functional.**reset_net**(*net: torch.nn.modules.module.Module*)

- *API in English*

参数 **net** –任何属于 `nn.Module` 子类的网络

返回 None

将网络的状态重置。做法是遍历网络中的所有 `Module`，若含有 `reset()` 函数，则调用。

- *中文 API*

参数 **net** –Any network inherits from `nn.Module`

返回 None

Reset the whole network. Walk through every `Module` and call their `reset()` function if exists.

spikingjelly.clock_driven.functional.**set_monitor**(*net: torch.nn.modules.module.Module, monitor_state*)

- *API in English*

参数

- **net** –任何属于 `nn.Module` 子类的网络

- **monitor_state** (*bool*) –表示开启或关闭 monitor

返回 None

将 net 中的所有含有监视器的模块，监视器状态设置为 monitor_state。

- *中文 API*

参数

- **net** –Any network inherits from nn.Module
- **monitor_state** (*bool*) –Indicating whether to turn on the monitor

返回 None

Set states of all monitors in modules of net to monitor_state.

spikingjelly.clock_driven.functional.**spike_cluster**(*v: torch.Tensor*, *v_threshold*, *T_in: int*)

- *API in English*

参数

- **v** –shape=[T, N]，N 个神经元在 t=[0, 1, ⋯, T-1] 时刻的电压值
- **v_threshold** (*float or tensor*) –神经元的阈值电压，float 或者是 shape=[N] 的 tensor
- **T_in** –脉冲聚类的距离阈值。一个脉冲聚类满足，内部任意 2 个相邻脉冲的距离不大于 T_in，而其内部任一脉冲与外部的脉冲距离大于 T_in。

返回

一个元组，包含

- **N_o** –shape=[N]，N 个神经元的输出脉冲的脉冲聚类的数量
- **k_positive** –shape=[N]，bool 类型的 tensor，索引。需要注意的是，k_positive 可能是一个全 False 的 tensor
- **k_negative** –shape=[N]，bool 类型的 tensor，索引。需要注意的是，k_negative 可能是一个全 False 的 tensor

返回类型 (Tensor, Tensor, Tensor)

STCA: Spatio-Temporal Credit Assignment with Delayed Feedback in Deep Spiking Neural Networks一文提出的脉冲聚类方法。如果想使用该文中定义的损失，可以参考如下代码：

```
v_k_negative = out_v * k_negative.float().sum(dim=0)
v_k_positive = out_v * k_positive.float().sum(dim=0)
loss0 = ((N_o > N_d).float() * (v_k_negative - 1.0)).sum()
```

(下页继续)

(续上页)

```
loss1 = ((N_o < N_d).float() * (1.0 - v_k_positive)).sum()
loss = loss0 + loss1
```

- *中文 API*

参数

- **v** –shape=[T, N], membrane potentials of N neurons when t=[0, 1, ⋯, T-1]
- **v_threshold** (*float or tensor*) –Threshold voltage(s) of the neurons, float or tensor of the shape=[N]
- **T_in** –Distance threshold of the spike clusters. A spike cluster satisfies that the distance of any two adjacent spikes within cluster is NOT greater than `T_in` and the distance between any internal and any external spike of cluster is greater than `T_in`.

返回

A tuple containing

- **N_o** –shape=[N], numbers of spike clusters of N neurons' output spikes
- **k_positive** –shape=[N], tensor of type BoolTensor, indexes. Note that k_positive can be a tensor filled with False
- **k_negative** –shape=[N], tensor of type BoolTensor, indexes. Note that k_negative can be a tensor filled with False

返回类型 (Tensor, Tensor, Tensor)

A spike clustering method proposed in STCA: Spatio-Temporal Credit Assignment with Delayed Feedback in Deep Spiking Neural Networks. You can refer to the following code if this form of loss function is needed:

```
v_k_negative = out_v * k_negative.float().sum(dim=0)
v_k_positive = out_v * k_positive.float().sum(dim=0)
loss0 = ((N_o > N_d).float() * (v_k_negative - 1.0)).sum()
loss1 = ((N_o < N_d).float() * (1.0 - v_k_positive)).sum()
loss = loss0 + loss1
```

spikingjelly.clock_driven.functional.**spike_similar_loss**(*spikes: torch.Tensor, labels: torch.Tensor, kernel_type='linear', loss_type='mse', *args*)

- *API in English*

参数

- **spikes** –shape=[N, M, T]，N 个数据生成的脉冲

- **labels** –shape=[N, C]，N 个数据的标签，`labels[i][k] == 1` 表示数据 i 属于第 k 类，反之亦然，允许多标签
- **kernel_type** (*str*) –使用内积来衡量两个脉冲之间的相似性，`kernel_type` 是计算内积时，所使用的核函数种类
- **loss_type** (*str*) –返回哪种损失，可以为’mse’，‘l1’，‘bce’
- **args** –用于计算内积的额外参数

返回 shape=[1] 的 tensor，相似损失

将 N 个数据输入到输出层有 M 个神经元的 SNN，运行 T 步，得到 shape=[N, M, T] 的脉冲。这 N 个数据的标签为 shape=[N, C] 的 `labels`。

用 shape=[N, N] 的矩阵 `sim` 表示**实际相似度矩阵**，`sim[i][j] == 1` 表示数据 i 与数据 j 相似，反之亦然。若 `labels[i]` 与 `labels[j]` 共享至少同一个标签，则认为他们相似，否则不相似。

用 shape=[N, N] 的矩阵 `sim_p` 表示**输出相似度矩阵**，`sim_p[i][j]` 的取值为 0 到 1，值越大表示数据 i 与数据 j 的脉冲越相似。

使用内积来衡量两个脉冲之间的相似性，`kernel_type` 是计算内积时，所使用的核函数种类：

- ‘linear’，线性内积，$\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \boldsymbol{x_i}^T \boldsymbol{y_j}$。
- ‘sigmoid’，Sigmoid 内积，$\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \text{sigmoid}(\alpha \boldsymbol{x_i}^T \boldsymbol{y_j})$，其中 $\alpha = args[0]$。
- ‘gaussian’，高斯内积，$\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \exp(-\frac{||\boldsymbol{x_i} - \boldsymbol{y_j}||^2}{2\sigma^2})$，其中 $\sigma = args[0]$。

当使用 Sigmoid 或高斯内积时，内积的取值范围均在 [0, 1] 之间；而使用线性内积时，为了保证内积取值仍然在 [0, 1] 之间，会进行归一化：按照

参数

- **spikes** –shape=[N, M, T], output spikes corresponding to a batch of N inputs
- **labels** –shape=[N, C], labels of inputs, `labels[i][k] == 1` means the i-th input belongs to the k-th category and vice versa. Multi-label input is allowed.
- **kernel_type** (*str*) –Type of kernel function used when calculating inner products. The inner product is the similarity measure of two spikes.
- **loss_type** (*str*) –Type of loss returned. Can be: ‘mse’，‘l1’，‘bce’
- **args** –Extra parameters for inner product

返回

shape=[1], similarity loss

A SNN consisting M neurons will receive a batch of N input data in each timestep (from 0 to T-1) and output a spike tensor of shape=[N, M, T]. The label is a tensor of shape=[N, C].

The **groundtruth similarity matrix** `sim` has a shape of [N, N]. `sim[i][j] == 1` indicates that input i is similar to input j and vice versa. If and only if `labels[i]` and `labels[j]` have at least one common label, they are viewed as similar.

The **output similarity matrix** `sim_p` has a shape of [N, N]. The value of `sim_p[i][j]` ranges from 0 to 1, represents the similarity between output spike from both input i and input j.

The similarity is measured by inner product of two spikes. `kernel_type` is the type of kernel function when calculating inner product:

- 'linear', Linear kernel, $\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \boldsymbol{x_i}^T \boldsymbol{y_j}$.
- 'sigmoid', Sigmoid kernel, $\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \text{sigmoid}(\alpha \boldsymbol{x_i}^T \boldsymbol{y_j})$, where $\alpha = args[0]$.
- 'gaussian', Gaussian kernel, $\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \exp(-\frac{||\boldsymbol{x_i} - \boldsymbol{y_j}||^2}{2\sigma^2})$, where $\sigma = args[0]$.

When Sigmoid or Gaussian kernel is applied, the inner product naturally lies in $[0, 1]$. To make the value consistent when using linear kernel, the result will be normalized as:

`spikingjelly.clock_driven.functional.`**`kernel_dot_product`**(*x: torch.Tensor, y: torch.Tensor, kernel='linear', *args*)

- *API in English*

参数

- **x** –shape=[N, M] 的 tensor，看作是 N 个 M 维向量
- **y** –shape=[N, M] 的 tensor，看作是 N 个 M 维向量
- **kernel** (*str*) –计算内积时所使用的核函数
- **args** –用于计算内积的额外的参数

返回 ret, shape=[N, N] 的 tensor，`ret[i][j]` 表示 `x[i]` 和 `y[j]` 的内积

计算批量数据 `x` 和 `y` 在核空间的内积。记 2 个 M 维 tensor 分别为 $\boldsymbol{x_i}$ 和 $\boldsymbol{y_j}$，`kernel` 定义了不同形式的内积：

- 'linear'，线性内积，$\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \boldsymbol{x_i}^T \boldsymbol{y_j}$。
- 'polynomial'，多项式内积，$\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = (\boldsymbol{x_i}^T \boldsymbol{y_j})^d$，其中 $d = args[0]$。
- 'sigmoid'，Sigmoid 内积，$\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \text{sigmoid}(\alpha \boldsymbol{x_i}^T \boldsymbol{y_j})$，其中 $\alpha = args[0]$。
- 'gaussian'，高斯内积，$\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \exp(-\frac{||\boldsymbol{x_i} - \boldsymbol{y_j}||^2}{2\sigma^2})$，其中 $\sigma = args[0]$。

- *中文 API*

参数

- **x** –Tensor of shape=[N, M]
- **y** –Tensor of shape=[N, M]
- **kernel** (*str*) –Type of kernel function used when calculating inner products.
- **args** –Extra parameters for inner product

返回 ret, Tensor of shape=[N, N], `ret[i][j]` is inner product of `x[i]` and `y[j]`.

Calculate inner product of `x` and `y` in kernel space. These 2 M-dim tensors are denoted by $\boldsymbol{x_i}$ and $\boldsymbol{y_j}$. `kernel` determine the kind of inner product:

- 'linear' –Linear kernel, $\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \boldsymbol{x_i}^T \boldsymbol{y_j}$.
- 'polynomial' –Polynomial kernel, $\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = (\boldsymbol{x_i}^T \boldsymbol{y_j})^d$, where $d = args[0]$.
- 'sigmoid' –Sigmoid kernel, $\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \text{sigmoid}(\alpha \boldsymbol{x_i}^T \boldsymbol{y_j})$, where $\alpha = args[0]$.
- 'gaussian' –Gaussian kernel, $\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \exp(-\frac{||\boldsymbol{x_i} - \boldsymbol{y_j}||^2}{2\sigma^2})$, where $\sigma = args[0]$.

`spikingjelly.clock_driven.functional.`**`set_threshold_margin`**(*output_layer: spikingjelly.clock_driven.neuron.BaseNode, label_one_hot: torch.Tensor, eval_threshold=1.0, threshold0=0.9, threshold1=1.1*)

- *API in English*

参数

- **`output_layer`** –用于分类的网络的输出层，输出层输出 shape=[batch_size, C]
- **`label_one_hot`** –one hot 格式的样本标签，shape=[batch_size, C]
- **`eval_threshold`** (*`float`*) –输出层神经元在测试（推理）时使用的电压阈值
- **`threshold0`** (*`float`*) –输出层神经元在训练时，负样本的电压阈值
- **`threshold1`** (*`float`*) –输出层神经元在训练时，正样本的电压阈值

返回 None

对于用来分类的网络，为输出层神经元的电压阈值设置一定的裕量，以获得更好的分类性能。

类别总数为 C，网络的输出层共有 C 个神经元。网络在训练时，当输入真实类别为 i 的数据，输出层中第 i 个神经元的电压阈值会被设置成 `threshold1`，而其他神经元的电压阈值会被设置成 `threshold0`。而在测试（推理）时，输出层中神经元的电压阈值被统一设置成 `eval_threshold`。

- *中文 API*

参数

- **`output_layer`** –The output layer of classification network, where the shape of output should be [batch_size, C]
- **`label_one_hot`** –Labels in one-hot format, shape=[batch_size, C]

- **eval_threshold** (*float*) –Voltage threshold of neurons in output layer when evaluating (inference)
- **threshold0** (*float*) –Voltage threshold of the corresponding neurons of **negative** samples in output layer when training
- **threshold1** (*float*) –Voltage threshold of the corresponding neurons of **positive** samples in output layer when training

返回 None

Set voltage threshold margin for neurons in the output layer to reach better performance in classification task.

When there are C different classes, the output layer contains C neurons. During training, when the input with groundtruth label i are sent into the network, the voltage threshold of the i-th neurons in the output layer will be set to `threshold1` and the remaining will be set to `threshold0`.

During inference, the voltage thresholds of **ALL** neurons in the output layer will be set to `eval_threshold`.

`spikingjelly.clock_driven.functional.`**`redundant_one_hot`**(*labels: torch.Tensor, num_classes: int, n: int*)

- *API in English*

参数

- **labels** –shape=[batch_size] 的 tensor，表示 `batch_size` 个标签
- **num_classes** (*int*) –类别总数
- **n** (*int*) –表示每个类别所用的编码数量

返回 shape=[batch_size, num_classes * n] 的 tensor

对数据进行冗余的 one-hot 编码，每一类用 `n` 个 1 和 `(num_classes - 1) * n` 个 0 来编码。

示例：

```
>>> num_classes = 3
>>> n = 2
>>> labels = torch.randint(0, num_classes, [4])
>>> labels
tensor([0, 1, 1, 0])
>>> codes = functional.redundant_one_hot(labels, num_classes, n)
>>> codes
tensor([[1., 1., 0., 0., 0., 0.],
        [0., 0., 1., 1., 0., 0.],
        [0., 0., 1., 1., 0., 0.],
        [1., 1., 0., 0., 0., 0.]])
```

- *中文 API*

参数

- **labels** –Tensor of shape=[batch_size], `batch_size` labels
- **num_classes** (*int*) –The total number of classes.
- **n** (*int*) –The encoding length for each class.

返回 Tensor of shape=[batch_size, num_classes * n]

Redundant one-hot encoding for data. Each class is encoded to `n` 1's and `(num_classes - 1) * n` 0's

e.g.:

```
>>> num_classes = 3
>>> n = 2
>>> labels = torch.randint(0, num_classes, [4])
>>> labels
tensor([0, 1, 1, 0])
>>> codes = functional.redundant_one_hot(labels, num_classes, n)
>>> codes
tensor([[1., 1., 0., 0., 0., 0.],
        [0., 0., 1., 1., 0., 0.],
        [0., 0., 1., 1., 0., 0.],
        [1., 1., 0., 0., 0., 0.]])
```

`spikingjelly.clock_driven.functional.`**`first_spike_index`**(*spikes: torch.Tensor*)

- *API in English*

参数 **spikes** –shape=[*, T]，表示任意个神经元在 t=0, 1, ···, T-1，共 T 个时刻的输出脉冲

返回 index, shape=[*, T]，为 `True` 的位置表示该神经元首次释放脉冲的时刻

输入若干个神经元的输出脉冲，返回一个与输入相同 shape 的 `bool` 类型的 index。index 为 `True` 的位置，表示该神经元首次释放脉冲的时刻。

示例：

```
>>> spikes = (torch.rand(size=[2, 3, 8]) >= 0.8).float()
>>> spikes
tensor([[[0., 0., 0., 0., 0., 0., 0., 0.],
 [1., 0., 0., 0., 0., 0., 1., 0.],
 [0., 1., 0., 0., 0., 1., 0., 1.]],

[[0., 0., 1., 1., 0., 0., 0., 1.],
 [1., 1., 0., 0., 1., 0., 0., 0.],
 [0., 0., 0., 1., 0., 0., 0., 0.]]])
>>> first_spike_index(spikes)
```

(下页继续)

(续上页)

```
tensor([[[False, False, False, False, False, False, False, False],
 [ True, False, False, False, False, False, False, False],
 [False,  True, False, False, False, False, False, False]],

[[False, False,  True, False, False, False, False, False],
 [ True, False, False, False, False, False, False, False],
 [False, False, False,  True, False, False, False, False]]])
```

- *中文 API*

参数 **spikes** –shape=[*, T], indicates the output spikes of some neurons when t=0, 1, ⋯, T-1.

返回 index, shape=[*, T], the index of `True` represents the moment of first spike.

Return an `index` tensor of the same shape of input tensor, which is the output spike of some neurons. The index of `True` represents the moment of first spike.

e.g.:

```
>>> spikes = (torch.rand(size=[2, 3, 8]) >= 0.8).float()
>>> spikes
tensor([[[0., 0., 0., 0., 0., 0., 0., 0.],
 [1., 0., 0., 0., 0., 0., 1., 0.],
 [0., 1., 0., 0., 0., 1., 0., 1.]],

[[0., 0., 1., 1., 0., 0., 0., 1.],
 [1., 1., 0., 0., 1., 0., 0., 0.],
 [0., 0., 0., 1., 0., 0., 0., 0.]]])
>>> first_spike_index(spikes)
tensor([[[False, False, False, False, False, False, False, False],
 [ True, False, False, False, False, False, False, False],
 [False,  True, False, False, False, False, False, False]],

[[False, False,  True, False, False, False, False, False],
 [ True, False, False, False, False, False, False, False],
 [False, False, False,  True, False, False, False, False]]])
```

`spikingjelly.clock_driven.functional.`**`spike_mse_loss`**(*x: torch.Tensor, spikes: torch.Tensor*)

- *API in English*

参数

- **x** –任意 tensor

- **spikes** –脉冲 tensor。要求 `spikes` 中的元素只能为 `0` 和 `1`，或只为 `False` 和 `True`，且 `spikes.shape` 必须与 `x.shape` 相同

返回 `x` 和 `spikes` 逐元素的均方误差（L2 范数的平方）

这个函数与 `torch.nn.functional.mse_loss()` 相比，针对脉冲数据进行了加速。其计算按照

$$(x-s)^2 = x^2 + s^2 - 2xs = x^2 + (1-2x)s$$

注解: 由于计算图已经改变，此函数计算出的梯度 $\frac{\partial L}{\partial s}$ 与 `torch.nn.functional.mse_loss()` 计算出的是不一样的。

- *中文 API*

参数

- **x** –an arbitrary tensor
- **spikes** –spiking tensor. The elements in `spikes` must be `0` and `1` or `False` and `True`, and `spikes.shape` should be same with `x.shape`

返回 the mean squared error (squared L2 norm) between each element in `x` and `spikes`

This function is faster than `torch.nn.functional.mse_loss()` for its optimization on spiking data. The compulation is carried out as

$$(x-s)^2 = x^2 + s^2 - 2xs = x^2 + (1-2x)s$$

Note

The computation graph of this function is different with the standard MSE. So $\frac{\partial L}{\partial s}$ compulated by this function is different with that by `torch.nn.functional.mse_loss()`.

---

spikingjelly.clock_driven.layer package

Module contents

class `spikingjelly.clock_driven.layer.`**`NeuNorm`**(*in_channels*, *height*, *width*, *k=0.9*, *shared_across_channels=False*)

基类: `torch.nn.modules.module.Module`

- *API in English*

参数

- **in_channels** –输入数据的通道数
- **height** –输入数据的宽
- **width** –输入数据的高
- **k** –动量项系数
- **shared_across_channels** –可学习的权重 `w` 是否在通道这一维度上共享。设置为 `True` 可以大幅度节省内存

Direct Training for Spiking Neural Networks: Faster, Larger, Better 中提出的 NeuNorm 层。NeuNorm 层必须放在二维卷积层后的脉冲神经元后，例如：

```
Conv2d -> LIF -> NeuNorm
```

要求输入的尺寸是 `[batch_size, in_channels, height, width]`。

`in_channels` 是输入到 NeuNorm 层的通道数，也就是论文中的 F。

`k` 是动量项系数，相当于论文中的 $k_{\tau 2}$。

论文中的 $\frac{v}{F}$ 会根据 $k_{\tau 2} + vF = 1$ 自动算出。

- *中文 API*

参数

- **in_channels** –channels of input
- **height** –height of input
- **width** –height of width
- **k** –momentum factor
- **shared_across_channels** –whether the learnable parameter `w` is shared over channel dim. If set `True`, the consumption of memory can decrease largely

The NeuNorm layer is proposed in Direct Training for Spiking Neural Networks: Faster, Larger, Better.

It should be placed after spiking neurons behind convolution layer, e.g.,

```
Conv2d -> LIF -> NeuNorm
```

The input should be a 4-D tensor with `shape = [batch_size, in_channels, height, width]`.

`in_channels` is the channels of input，which is F in the paper.

`k` is the momentum factor，which is $k_{\tau 2}$ in the paper.

$\frac{v}{F}$ will be calculated by $k_{\tau 2} + vF = 1$ autonomously.

forward(*in_spikes: torch.Tensor*)

reset()

- *API in English*

返回 None

本层是一个有状态的层。此函数重置本层的状态变量。

- *中文 API*

返回 None

This layer is stateful. This function will reset all stateful variables.

training: bool

class spikingjelly.clock_driven.layer.**DCT**(*kernel_size*)

基类: torch.nn.modules.module.Module

- *API in English*

参数 **kernel_size** –进行分块 DCT 变换的块大小

将输入的 shape = [*, W, H] 的数据进行分块 DCT 变换的层，* 表示任意额外添加的维度。变换只在最后 2 维进行，要求 W 和 H 都能整除 kernel_size。

DCT 是 AXAT 的一种特例。

- *中文 API*

参数 **kernel_size** –block size for DCT transform

Apply Discrete Cosine Transform on input with shape = [*, W, H], where * means any number of additional dimensions. DCT will only applied in the last two dimensions. W and H should be divisible by kernel_size.

Note that DCT is a special case of AXAT.

forward(*x: torch.Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**AXAT**(*in_features*, *out_features*)

基类: torch.nn.modules.module.Module

- *API in English*

参数

- **in_features** –输入数据的最后 2 维的尺寸。输入应该是 shape = [*, in_features, in_features]

- **out_features** –输出数据的最后 2 维的尺寸。输出数据为 `shape = [*, out_features, out_features]`

对输入数据 X 在最后 2 维进行线性变换 AXA^T 的操作，A 是 `shape = [out_features, in_features]` 的矩阵。

将输入的数据看作是批量个 `shape = [in_features, in_features]` 的矩阵.

- *中文 API*

参数

- **in_features** –feature number of input at last two dimensions. The input should be `shape = [*, in_features, in_features]`
- **out_features** –feature number of output at last two dimensions. The output will be `shape = [*, out_features, out_features]`

Apply AXA^T transform on input X at the last two dimensions. A is a tensor with `shape = [out_features, in_features]`.

The input will be regarded as a batch of tensors with `shape = [in_features, in_features]`.

forward(*x: torch.Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**Dropout**(*p=0.5*)

基类：`torch.nn.modules.module.Module`

- API in English

参数 **p** (*float*) –每个元素被设置为 0 的概率

与 `torch.nn.Dropout` 的几乎相同。区别在于，在每一轮的仿真中，被设置成 0 的位置不会发生改变；直到下一轮运行，即网络调用 reset() 函数后，才会按照概率去重新决定，哪些位置被置 0。

小技巧: 这种 Dropout 最早由 Enabling Spike-based Backpropagation for Training Deep Neural Network Architectures 一文进行详细论述：

There is a subtle difference in the way dropout is applied in SNNs compared to ANNs. In ANNs, each epoch of training has several iterations of mini-batches. In each iteration, randomly selected units (with dropout ratio of p) are disconnected from the network while weighting by its posterior probability $(1-p)$. However, in SNNs, each iteration has more than one forward propagation depending on the time length of the spike train. We back-propagate the output error and modify the network parameters only at the last time step. For dropout to be effective in our training method, it has to be ensured that the set of connected units within an iteration of mini-batch data is not changed, such that the neural network is constituted by the same random subset of units during each forward propagation within a single iteration. On the other hand, if the units are randomly connected at each time-step,

the effect of dropout will be averaged out over the entire forward propagation time within an iteration. Then, the dropout effect would fade-out once the output error is propagated backward and the parameters are updated at the last time step. Therefore, we need to keep the set of randomly connected units for the entire time window within an iteration.

- 中文 API

参数 **p** (*float*) –probability of an element to be zeroed

This layer is almost same with `torch.nn.Dropout`. The difference is that elements have been zeroed at first step during a simulation will always be zero. The indexes of zeroed elements will be update only after `reset()` has been called and a new simulation is started.

Tip

This kind of Dropout is firstly described in Enabling Spike-based Backpropagation for Training Deep Neural Network Architectures:

There is a subtle difference in the way dropout is applied in SNNs compared to ANNs. In ANNs, each epoch of training has several iterations of mini-batches. In each iteration, randomly selected units (with dropout ratio of p) are disconnected from the network while weighting by its posterior probability $(1-p)$. However, in SNNs, each iteration has more than one forward propagation depending on the time length of the spike train. We back-propagate the output error and modify the network parameters only at the last time step. For dropout to be effective in our training method, it has to be ensured that the set of connected units within an iteration of mini-batch data is not changed, such that the neural network is constituted by the same random subset of units during each forward propagation within a single iteration. On the other hand, if the units are randomly connected at each time-step, the effect of dropout will be averaged out over the entire forward propagation time within an iteration. Then, the dropout effect would fade-out once the output error is propagated backward and the parameters are updated at the last time step. Therefore, we need to keep the set of randomly connected units for the entire time window within an iteration.

---

extra_repr()

create_mask(*x: torch.Tensor*)

forward(*x: torch.Tensor*)

reset()

- *API in English*

返回 None

本层是一个有状态的层。此函数重置本层的状态变量。

- 中文 API

返回 None

This layer is stateful. This function will reset all stateful variables.

training: bool

class spikingjelly.clock_driven.layer.**Dropout2d**(*p=0.2*)

基类：*spikingjelly.clock_driven.layer.Dropout*

- API in English

参数 **p** (*float*) –每个元素被设置为 0 的概率

与 `torch.nn.Dropout2d` 的几乎相同。区别在于，在每一轮的仿真中，被设置成 0 的位置不会发生改变；直到下一轮运行，即网络调用 reset() 函数后，才会按照概率去重新决定，哪些位置被置 0。

关于 SNN 中 Dropout 的更多信息，参见 layer.Dropout。

- 中文 API

参数 **p** (*float*) –probability of an element to be zeroed

This layer is almost same with `torch.nn.Dropout2d`. The difference is that elements have been zeroed at first step during a simulation will always be zero. The indexes of zeroed elements will be update only after `reset()` has been called and a new simulation is started.

For more information about Dropout in SNN, refer to layer.Dropout.

create_mask(*x: torch.Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**MultiStepDropout**(*p=0.5*)

基类：*spikingjelly.clock_driven.layer.Dropout*

- API in English

参数 **p** (*float*) –每个元素被设置为 0 的概率

与 `torch.nn.Dropout` 的几乎相同。区别在于，在每一轮的仿真中，被设置成 0 的位置不会发生改变；直到下一轮运行，即网络调用 reset() 函数后，才会按照概率去重新决定，哪些位置被置 0。

小技巧: 这种 Dropout 最早由 Enabling Spike-based Backpropagation for Training Deep Neural Network Architectures 一文进行详细论述：

There is a subtle difference in the way dropout is applied in SNNs compared to ANNs. In ANNs, each epoch of training has several iterations of mini-batches. In each iteration, randomly selected units (with dropout ratio of p) are disconnected from the network while weighting by its posterior probability $(1-p)$. However, in SNNs,

each iteration has more than one forward propagation depending on the time length of the spike train. We back-propagate the output error and modify the network parameters only at the last time step. For dropout to be effective in our training method, it has to be ensured that the set of connected units within an iteration of mini-batch data is not changed, such that the neural network is constituted by the same random subset of units during each forward propagation within a single iteration. On the other hand, if the units are randomly connected at each time-step, the effect of dropout will be averaged out over the entire forward propagation time within an iteration. Then, the dropout effect would fade-out once the output error is propagated backward and the parameters are updated at the last time step. Therefore, we need to keep the set of randomly connected units for the entire time window within an iteration.

- 中文 API

参数 **p** (*float*) –probability of an element to be zeroed

This layer is almost same with `torch.nn.Dropout`. The difference is that elements have been zeroed at first step during a simulation will always be zero. The indexes of zeroed elements will be update only after `reset()` has been called and a new simulation is started.

Tip

This kind of Dropout is firstly described in Enabling Spike-based Backpropagation for Training Deep Neural Network Architectures:

There is a subtle difference in the way dropout is applied in SNNs compared to ANNs. In ANNs, each epoch of training has several iterations of mini-batches. In each iteration, randomly selected units (with dropout ratio of p) are disconnected from the network while weighting by its posterior probability $(1-p)$. However, in SNNs, each iteration has more than one forward propagation depending on the time length of the spike train. We back-propagate the output error and modify the network parameters only at the last time step. For dropout to be effective in our training method, it has to be ensured that the set of connected units within an iteration of mini-batch data is not changed, such that the neural network is constituted by the same random subset of units during each forward propagation within a single iteration. On the other hand, if the units are randomly connected at each time-step, the effect of dropout will be averaged out over the entire forward propagation time within an iteration. Then, the dropout effect would fade-out once the output error is propagated backward and the parameters are updated at the last time step. Therefore, we need to keep the set of randomly connected units for the entire time window within an iteration.

---

forward(*x_seq: torch.Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**MultiStepDropout2d**(*p=0.2*)

基类: *spikingjelly.clock_driven.layer.Dropout2d*

- API in English

  参数 **p** (*float*) –每个元素被设置为 0 的概率

与 `torch.nn.Dropout2d` 的几乎相同。区别在于，在每一轮的仿真中，被设置成 0 的位置不会发生改变；直到下一轮运行，即网络调用 reset() 函数后，才会按照概率去重新决定，哪些位置被置 0。

关于 SNN 中 Dropout 的更多信息，参见 layer.Dropout。

- 中文 API

  参数 **p** (*float*) –probability of an element to be zeroed

This layer is almost same with `torch.nn.Dropout2d`. The difference is that elements have been zeroed at first step during a simulation will always be zero. The indexes of zeroed elements will be update only after `reset()` has been called and a new simulation is started.

For more information about Dropout in SNN, refer to layer.Dropout.

forward(*x_seq: torch.Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**SynapseFilter**(*tau=100.0*, *learnable=False*)

基类：`torch.nn.modules.module.Module`

- *API in English*

  参数

  - **tau** –time 突触上电流衰减的时间常数
  - **learnable** –时间常数在训练过程中是否是可学习的。若为 `True`，则 `tau` 会被设定成时间常数的初始值

具有滤波性质的突触。突触的输出电流满足，当没有脉冲输入时，输出电流指数衰减：

$$\tau \frac{\mathrm{d}I(t)}{\mathrm{d}t} = -I(t)$$

当有新脉冲输入时，输出电流自增 1：

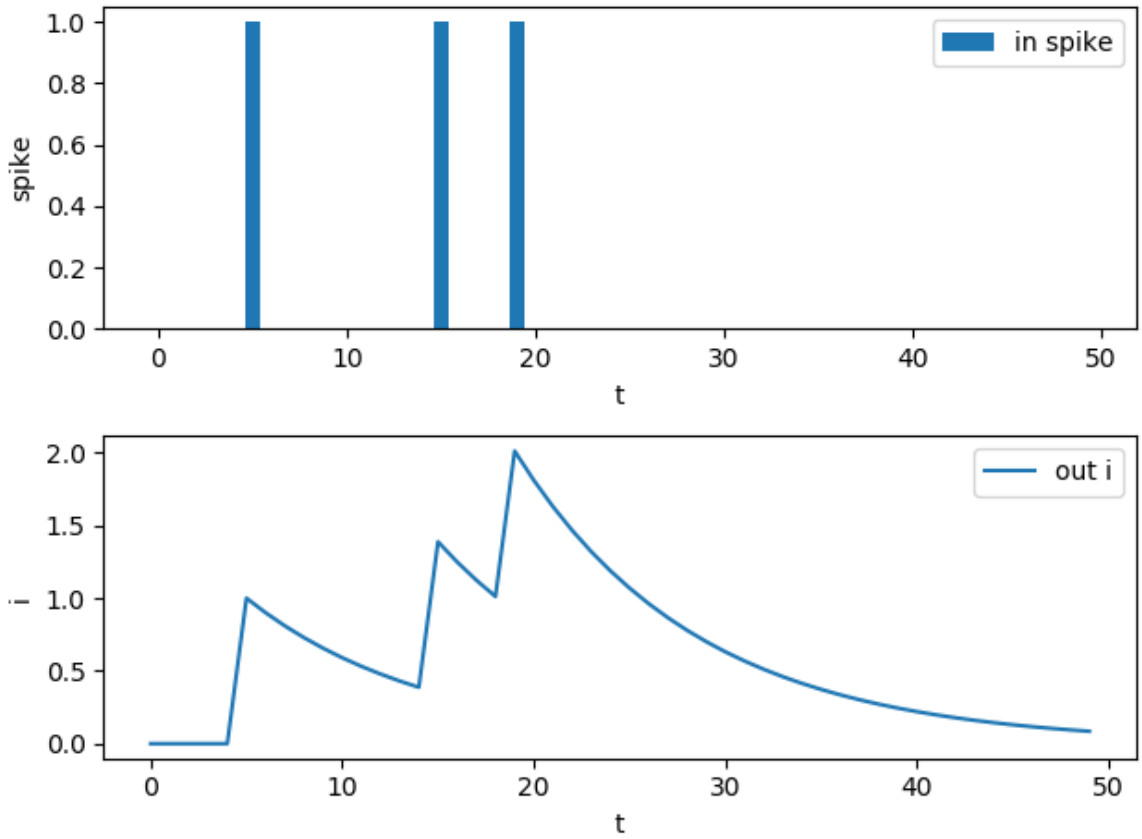
$$I(t) = I(t) + 1$$

记输入脉冲为 $S(t)$，则离散化后，统一的电流更新方程为：

$$I(t) = I(t-1) - (1 - S(t))\frac{1}{\tau}I(t-1) + S(t)$$

这种突触能将输入脉冲进行平滑，简单的示例代码和输出结果：

```
T = 50
in_spikes = (torch.rand(size=[T]) >= 0.95).float()
lp_syn = LowPassSynapse(tau=10.0)
pyplot.subplot(2, 1, 1)
pyplot.bar(torch.arange(0, T).tolist(), in_spikes, label='in spike')
pyplot.xlabel('t')
pyplot.ylabel('spike')
pyplot.legend()

out_i = []
for i in range(T):
    out_i.append(lp_syn(in_spikes[i]))
pyplot.subplot(2, 1, 2)
pyplot.plot(out_i, label='out i')
pyplot.xlabel('t')
pyplot.ylabel('i')
pyplot.legend()
pyplot.show()
```



输出电流不仅取决于当前时刻的输入，还取决于之前的输入，使得该突触具有了一定的记忆能力。

这种突触偶有使用，例如：

Unsupervised learning of digit recognition using spike-timing-dependent plasticity

Exploiting Neuron and Synapse Filter Dynamics in Spatial Temporal Learning of Deep Spiking Neural Network

另一种视角是将其视为一种输入为脉冲，并输出其电压的 LIF 神经元。并且该神经元的发放阈值为 $+\infty$ 。

神经元最后累计的电压值一定程度上反映了该神经元在整个仿真过程中接收脉冲的数量，从而替代了传统的直接对输出脉冲计数（即发放频率）来表示神经元活跃程度的方法。因此通常用于最后一层，在以下文章中使用：

Enabling spike-based backpropagation for training deep neural network architectures

- *中文 API*

参数

- **tau** –time constant that determines the decay rate of current in the synapse
- **learnable** –whether time constant is learnable during training. If `True`, then `tau` will be the initial value of time constant

The synapse filter that can filter input current. The output current will decay when there is no input spike:

$$\tau \frac{\mathrm{d}I(t)}{\mathrm{d}t} = -I(t)$$

The output current will increase 1 when there is a new input spike:

$$I(t) = I(t) + 1$$

Denote the input spike as $S(t)$, then the discrete current update equation is as followed:

$$I(t) = I(t-1) - (1 - S(t))\frac{1}{\tau}I(t-1) + S(t)$$

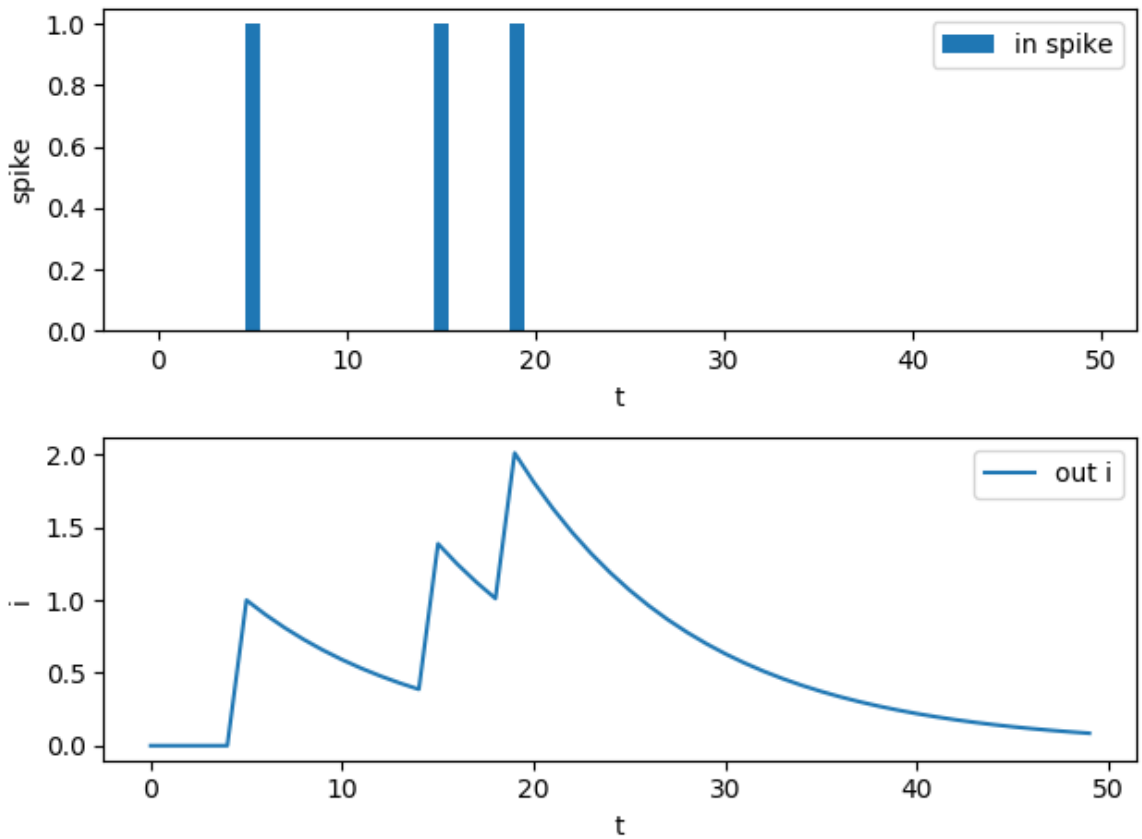
This synapse can smooth input. Here is the example and output:

```
T = 50
in_spikes = (torch.rand(size=[T]) >= 0.95).float()
lp_syn = LowPassSynapse(tau=10.0)
pyplot.subplot(2, 1, 1)
pyplot.bar(torch.arange(0, T).tolist(), in_spikes, label='in spike')
pyplot.xlabel('t')
pyplot.ylabel('spike')
pyplot.legend()

out_i = []
for i in range(T):
```

(下页继续)

(续上页)

```
    out_i.append(lp_syn(in_spikes[i]))
pyplot.subplot(2, 1, 2)
pyplot.plot(out_i, label='out i')
pyplot.xlabel('t')
pyplot.ylabel('i')
pyplot.legend()
pyplot.show()
```



The output current is not only determined by the present input but also by the previous input, which makes this synapse have memory.

This synapse is sometimes used, e.g.:

Unsupervised learning of digit recognition using spike-timing-dependent plasticity

Exploiting Neuron and Synapse Filter Dynamics in Spatial Temporal Learning of Deep Spiking Neural Network

Another view is regarding this synapse as a LIF neuron with a $+\infty$ threshold voltage.

The final output of this synapse (or the final voltage of this LIF neuron) represents the accumulation of input spikes, which substitute for traditional firing rate that indicates the excitatory level. So, it can be used in the last layer of the network, e.g.:

Enabling spike-based backpropagation for training deep neural network architectures

extra_repr()

forward(*in_spikes: torch.Tensor*)

reset()

- *API in English*

  返回 None

本层是一个有状态的层。此函数重置本层的状态变量。

- *中文 API*

  返回 None

This layer is stateful. This function will reset all stateful variables.

training: bool

class spikingjelly.clock_driven.layer.**ChannelsPool**(*pool: torch.nn.modules.pooling.MaxPool1d*)

基类：torch.nn.modules.module.Module

- *API in English*

  参数 **pool** –nn.MaxPool1d 或 nn.AvgPool1d，池化层

使用 pool 将输入的 4-D 数据在第 1 个维度上进行池化。

示例代码：

```
>>> cp = ChannelsPool(torch.nn.MaxPool1d(2, 2))
>>> x = torch.rand(size=[2, 8, 4, 4])
>>> y = cp(x)
>>> y.shape
torch.Size([2, 4, 4, 4])
```

- *中文 API*

  参数 **pool** –nn.MaxPool1d or nn.AvgPool1d, the pool layer

Use pool to pooling 4-D input at dimension 1.

Examples:

```
>>> cmp = ChannelsPool(torch.nn.MaxPool1d(2, 2))
>>> x = torch.rand(size=[2, 8, 4, 4])
>>> y = cp(x)
>>> y.shape
torch.Size([2, 4, 4, 4])
```

forward(*x: torch.Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**DropConnectLinear**(*in_features: int, out_features: int, bias: bool = True, p: float = 0.5, samples_num: int = 1024, invariant: bool = False, activation: torch.nn.modules.module.Module = ReLU()*)

基类：torch.nn.modules.module.Module

- *API in English*

参数

- **in_features** (*int*) –每个输入样本的特征数
- **out_features** (*int*) –每个输出样本的特征数
- **bias** (*bool*) –若为 False，则本层不会有可学习的偏置项。默认为 True
- **p** (*float*) –每个连接被断开的概率。默认为 0.5
- **samples_num** (*int*) –在推理时，从高斯分布中采样的数据数量。默认为 1024
- **invariant** (*bool*) –若为 True，线性层会在第一次执行前向传播时被按概率断开，断开后的线性层会保持不变，直到 reset() 函数被调用，线性层恢复为完全连接的状态。完全连接的线性层，调用 reset() 函数后的第一次前向传播时被重新按概率断开。若为 False，在每一次前向传播时线性层都会被重新完全连接再按概率断开。阅读 layer.Dropout 以获得更多关于此参数的信息。默认为 False
- **activation** (*None or nn.Module*) –在线性层后的激活层

DropConnect，由 Regularization of Neural Networks using DropConnect 一文提出。DropConnect 与 Dropout 非常类似，区别在于 DropConnect 是以概率 p 断开连接，而 Dropout 是将输入以概率置 0。

注解: 在使用 DropConnect 进行推理时，输出的 tensor 中的每个元素，都是先从高斯分布中采样，通过激活层激活，再在采样数量上进行平均得到的。详细的流程可以在 Regularization of Neural Networks

using DropConnect 一文中的 *Algorithm 2* 找到。激活层 `activation` 在中间的步骤起作用，因此我们将其作为模块的成员。

- *中文 API*

参数

- **in_features** (*int*) –size of each input sample
- **out_features** (*int*) –size of each output sample
- **bias** (*bool*) –If set to `False`, the layer will not learn an additive bias. Default: `True`
- **p** (*float*) –probability of an connection to be zeroed. Default: 0.5
- **samples_num** (*int*) –number of samples drawn from the Gaussian during inference. Default: 1024
- **invariant** (*bool*) –If set to `True`, the connections will be dropped at the first time of forward and the dropped connections will remain unchanged until `reset()` is called and the connections recovery to fully-connected status. Then the connections will be re-dropped at the first time of forward after `reset()`. If set to `False`, the connections will be re-dropped at every forward. See layer.Dropout for more information to understand this parameter. Default: `False`
- **activation** (*None or nn.Module*) –the activation layer after the linear layer

DropConnect, which is proposed by Regularization of Neural Networks using DropConnect, is similar with Dropout but drop connections of a linear layer rather than the elements of the input tensor with probability `p`.

Note

When inference with DropConnect, every elements of the output tensor are sampled from a Gaussian distribution, activated by the activation layer and averaged over the sample number `samples_num`. See *Algorithm 2* in Regularization of Neural Networks using DropConnect for more details. Note that activation is an intermediate process. This is the reason why we include `activation` as a member variable of this module.

---

reset_parameters() → None

- *API in English*

返回 None

返回类型 None

初始化模型中的可学习参数。

- 中文 API

返回 None

返回类型 None

Initialize the learnable parameters of this module.

reset()

- API in English

返回 None

返回类型 None

将线性层重置为完全连接的状态，若 `self.activation` 也是一个有状态的层，则将其也重置。

- 中文 API

返回 None

返回类型 None

Reset the linear layer to fully-connected status. If `self.activation` is also stateful, this function will also reset it.

drop(*batch_size: int*)

forward(*input: torch.Tensor*) → torch.Tensor

extra_repr() → str

training: bool

class spikingjelly.clock_driven.layer.**MultiStepContainer**(**args*)

基类：`torch.nn.modules.module.Module`

forward(*x_seq: torch.Tensor*)

参数 **x_seq** (*torch.Tensor*) –shape=[T, batch_size, ⋯]

返回 y_seq, shape=[T, batch_size, ⋯]

返回类型 torch.Tensor

reset()

training: bool

class spikingjelly.clock_driven.layer.**SeqToANNContainer**(**args*)

基类：`torch.nn.modules.module.Module`

- API in English

参数 ***args** –无状态的单个或多个 ANN 网络层

包装无状态的 ANN 以处理序列数据的包装器。`shape=[T, batch_size, ...]` 的输入会被拼接成 `shape=[T * batch_size, ...]` 再送入被包装的模块。输出结果会被再拆成 `shape=[T, batch_size, ...]`。

示例代码

- *中文 API*

参数 ***args** –one or many stateless ANN layers

A container that contain sataeless ANN to handle sequential data. This container will concatenate inputs `shape=[T, batch_size, ...]` at time dimension as `shape=[T * batch_size, ...]`, and send the reshaped inputs to contained ANN. The output will be split to `shape=[T, batch_size, ...]`.

Examples:

forward(*x_seq: torch.Tensor*)

参数 **x_seq** (*torch.Tensor*) –shape=[T, batch_size, ⋯]

返回 y_seq, shape=[T, batch_size, ⋯]

返回类型 torch.Tensor

training: bool

class spikingjelly.clock_driven.layer.**STDPLearner**(*tau_pre: float, tau_post: float, f_pre, f_post*)

基类：`torch.nn.modules.module.Module`

```
import torch
import torch.nn as nn
from spikingjelly.clock_driven import layer, neuron, functional
from matplotlib import pyplot as plt
import numpy as np
def f_pre(x):
    return x.abs() + 0.1

def f_post(x):
    return - f_pre(x)

fc = nn.Linear(1, 1, bias=False)

stdp_learner = layer.STDPLearner(100., 100., f_pre, f_post)
trace_pre = []
trace_post = []
w = []
```

(下页继续)

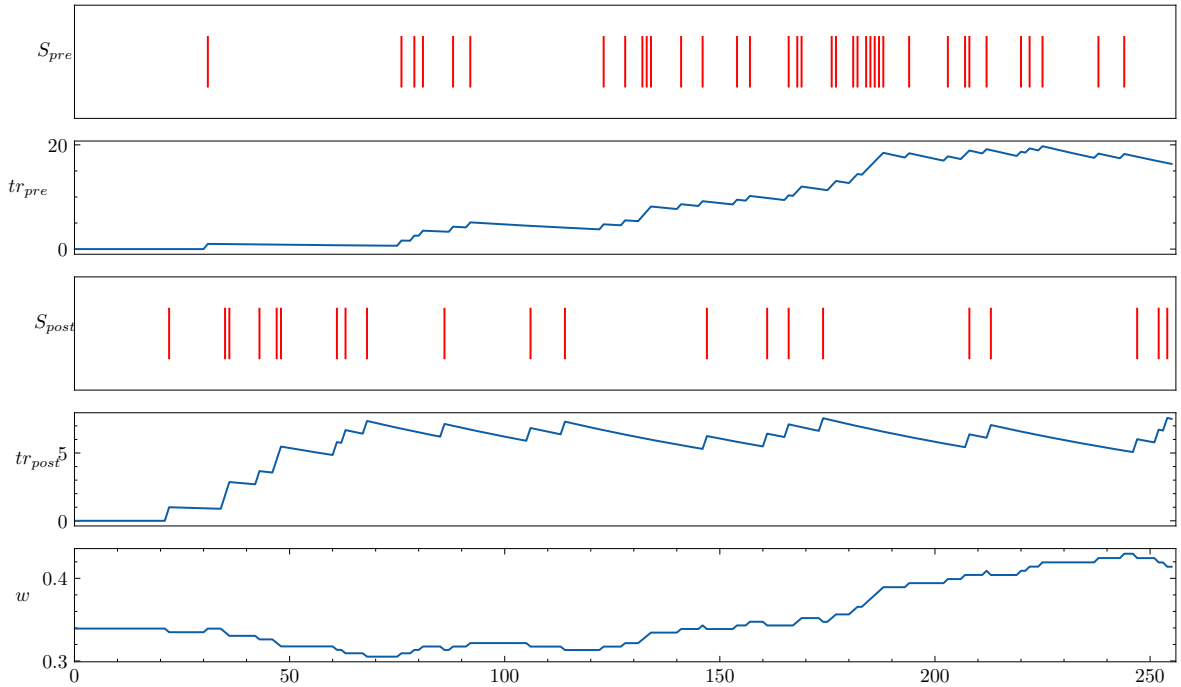
(续上页)

```
T = 256
s_pre = torch.zeros([T, 1])
s_post = torch.zeros([T, 1])
s_pre[0: T // 2] = (torch.rand_like(s_pre[0: T // 2]) > 0.95).float()
s_post[0: T // 2] = (torch.rand_like(s_post[0: T // 2]) > 0.9).float()

s_pre[T // 2:] = (torch.rand_like(s_pre[T // 2:]) > 0.8).float()
s_post[T // 2:] = (torch.rand_like(s_post[T // 2:]) > 0.95).float()

for t in range(T):
    stdp_learner.stdp(s_pre[t], s_post[t], fc, 1e-2)
    trace_pre.append(stdp_learner.trace_pre.item())
    trace_post.append(stdp_learner.trace_post.item())
    w.append(fc.weight.item())

plt.style.use('science')
fig = plt.figure(figsize=(10, 6))
s_pre = s_pre[:, 0].numpy()
s_post = s_post[:, 0].numpy()
t = np.arange(0, T)
plt.subplot(5, 1, 1)
plt.eventplot((t * s_pre)[s_pre == 1.], lineoffsets=0, colors='r')
plt.yticks([])
plt.ylabel('$S_{pre}$', rotation=0, labelpad=10)
plt.xticks([])
plt.xlim(0, T)
plt.subplot(5, 1, 2)
plt.plot(t, trace_pre)
plt.ylabel('$tr_{pre}$', rotation=0, labelpad=10)
plt.xticks([])
plt.xlim(0, T)

plt.subplot(5, 1, 3)
plt.eventplot((t * s_post)[s_post == 1.], lineoffsets=0, colors='r')
plt.yticks([])
plt.ylabel('$S_{post}$', rotation=0, labelpad=10)
plt.xticks([])
plt.xlim(0, T)
plt.subplot(5, 1, 4)
plt.plot(t, trace_post)
plt.ylabel('$tr_{post}$', rotation=0, labelpad=10)
plt.xticks([])
plt.xlim(0, T)
```

(下页继续)

(续上页)

```
plt.subplot(5, 1, 5)
plt.plot(t, w)
plt.ylabel('$w$', rotation=0, labelpad=10)
plt.xlim(0, T)

plt.show()
```



reset()

stdp(*s_pre: torch.Tensor*, *s_post: torch.Tensor*, *module: torch.nn.modules.module.Module*, *learning_rate: float*)

training: bool

class spikingjelly.clock_driven.layer.**PrintShapeModule**(*ext_str='PrintShapeModule'*)

基类：torch.nn.modules.module.Module

- *API in English*

  参数 **ext_str**(*str*) –额外打印的字符串

只打印 ext_str 和输入的 shape，不进行任何操作的网络层，可以用于 debug。

- *中文 API*

  参数 **ext_str**(*str*) –extra strings for printing

This layer will not do any operation but print `ext_str` and the shape of input, which can be used for debugging.

training: bool

forward(*x: torch.Tensor*)

spikingjelly.clock_driven.neuron package

Module contents

class spikingjelly.clock_driven.neuron.**BaseNode**(*v_threshold=1.0, v_reset=0.0, surrogate_function=Sigmoid(alpha=1.0, spiking=True), detach_reset=False, monitor_state=False*)

基类：`torch.nn.modules.module.Module`

- *API in English*

参数

- **v_threshold** –神经元的阈值电压
- **v_reset** –神经元的重置电压。如果不为 `None`，当神经元释放脉冲后，电压会被重置为 `v_reset`；如果设置为 `None`，则电压会被减去 `v_threshold`
- **surrogate_function** –反向传播时用来计算脉冲函数梯度的替代函数
- **detach_reset** –是否将 reset 过程的计算图分离
- **monitor_state** –是否设置监视器来保存神经元的电压和释放的脉冲。若为 `True`，则 `self.monitor` 是一个字典，键包括 `h, v s`，分别记录充电后的电压、释放脉冲后的电压、释放的脉冲。对应的值是一个链表。为了节省显存（内存），列表中存入的是原始变量转换为 `numpy` 数组后的值。还需要注意，`self.reset()` 函数会清空这些链表

可微分 SNN 神经元的基类神经元。

- *中文 API*

参数

- **v_threshold** –threshold voltage of neurons
- **v_reset** –reset voltage of neurons. If not `None`, voltage of neurons that just fired spikes will be set to `v_reset`. If `None`, voltage of neurons that just fired spikes will subtract `v_threshold`
- **surrogate_function** –surrogate function for replacing gradient of spiking functions during back-propagation

- **detach_reset** –whether detach the computation graph of reset
- **detach_reset** –whether detach the computation graph of reset
- **monitor_state** –whether to set a monitor to recode voltage and spikes of neurons. If `True`, `self.monitor` will be a dictionary with key `h` for recording membrane potential after charging, `v` for recording membrane potential after firing and `s` for recording output spikes. And the value of the dictionary is lists. To save memory, the elements in lists are `numpy` array converted from origin data. Besides, `self.reset()` will clear these lists in the dictionary

This class is the base class of differentiable spiking neurons.

abstract neuronal_charge(*dv: torch.Tensor*)

- *API in English*

定义神经元的充电差分方程。子类必须实现这个函数。

- *中文 API*

Define the charge difference equation. The sub-class must implement this function.

neuronal_fire()

- *API in English*

根据当前神经元的电压、阈值，计算输出脉冲。

- *中文 API*

Calculate out spikes of neurons by their current membrane potential and threshold voltage.

neuronal_reset()

- *API in English*

根据当前神经元释放的脉冲，对膜电位进行重置。

- *中文 API*

Reset the membrane potential according to neurons' output spikes.

extra_repr()

set_monitor(*monitor_state=True*)

- *API in English*

参数 **monitor_state** –`True` 或 `False`，表示开启或关闭 monitor

返回 None

设置开启或关闭 monitor。

- *中文 API*

参数 **monitor_state** –`True` or `False`, which indicates turn on or turn off the monitor

返回 None

Turn on or turn off the monitor.

forward(*dv: torch.Tensor*)

- *API in English*

参数 **dv** –输入到神经元的电压增量

返回 神经元的输出脉冲

按照充电、放电、重置的顺序进行前向传播。

- *中文 API*

参数 **dv** –increment of voltage inputted to neurons

返回 out spikes of neurons

Forward by the order of *neuronal_charge*, *neuronal_fire*, and *neuronal_reset*.

reset()

- *API in English*

返回 None

重置神经元为初始状态，也就是将电压设置为 `v_reset`。如果子类的神经元还含有其他状态变量，需要在此函数中将这些状态变量全部重置。

- *中文 API*

返回 None

Reset neurons to initial states, which means that set voltage to `v_reset`. Note that if the subclass has other stateful variables, these variables should be reset by this function.

training: bool

class spikingjelly.clock_driven.neuron.**IFNode**(*v_threshold=1.0, v_reset=0.0, surrogate_function=Sigmoid(alpha=1.0, spiking=True), detach_reset=False, monitor_state=False*)

基类：*spikingjelly.clock_driven.neuron.BaseNode*

- *API in English*

参数

- **v_threshold** –神经元的阈值电压
- **v_reset** –神经元的重置电压。如果不为 `None`，当神经元释放脉冲后，电压会被重置为 `v_reset`；如果设置为 `None`，则电压会被减去 `v_threshold`
- **surrogate_function** –反向传播时用来计算脉冲函数梯度的替代函数
- **detach_reset** –是否将 reset 过程的计算图分离
- **monitor_state** –是否设置监视器来保存神经元的电压和释放的脉冲。若为 `True`，则 `self.monitor` 是一个字典，键包括 `h`, `v` `s`，分别记录充电后的电压、释放脉冲后的电压、释放的脉冲。对应的值是一个链表。为了节省显存（内存），列表中存入的是原始变量转换为 `numpy` 数组后的值。还需要注意，`self.reset()` 函数会清空这些链表

Integrate-and-Fire 神经元模型，可以看作理想积分器，无输入时电压保持恒定，不会像 LIF 神经元那样衰减。其阈下神经动力学方程为：

$$\frac{\mathrm{d}V(t)}{\mathrm{d}t} = R_m I(t)$$

- *中文 API*

参数

- **v_threshold** –threshold voltage of neurons
- **v_reset** –reset voltage of neurons. If not `None`, voltage of neurons that just fired spikes will be set to `v_reset`. If `None`, voltage of neurons that just fired spikes will subtract `v_threshold`
- **surrogate_function** –surrogate function for replacing gradient of spiking functions during back-propagation
- **detach_reset** –whether detach the computation graph of reset
- **monitor_state** –whether to set a monitor to recode voltage and spikes of neurons. If `True`, `self.monitor` will be a dictionary with key `h` for recording membrane potential after charging, `v` for recording membrane potential after firing and `s` for recording output spikes. And the value of the dictionary is lists. To save memory, the elements in lists are `numpy` array converted from origin data. Besides, `self.reset()` will clear these lists in the dictionary

The Integrate-and-Fire neuron, which can be seen as a ideal integrator. The voltage of the IF neuron will not decay as that of the LIF neuron. The subthreshold neural dynamics of it is as followed:

$$\frac{\mathrm{d}V(t)}{\mathrm{d}t} = R_m I(t)$$

neuronal_charge(*dv: torch.Tensor*)

training: bool

class spikingjelly.clock_driven.neuron.**LIFNode**(*tau=100.0*, *v_threshold=1.0*, *v_reset=0.0*, *surrogate_function=Sigmoid(alpha=1.0, spiking=True)*, *detach_reset=False*, *monitor_state=False*)

基类：*spikingjelly.clock_driven.neuron.BaseNode*

- *API in English*

参数

- **tau** –膜电位时间常数。tau 对于这一层的所有神经元都是共享的
- **v_threshold** –神经元的阈值电压
- **v_reset** –神经元的重置电压。如果不为 None，当神经元释放脉冲后，电压会被重置为 v_reset；如果设置为 None，则电压会被减去 v_threshold
- **surrogate_function** –反向传播时用来计算脉冲函数梯度的替代函数
- **detach_reset** –是否将 reset 过程的计算图分离
- **monitor_state** –是否设置监视器来保存神经元的电压和释放的脉冲。若为 True，则 self.monitor 是一个字典，键包括 h, v s，分别记录充电后的电压、释放脉冲后的电压、释放的脉冲。对应的值是一个链表。为了节省显存（内存），列表中存入的是原始变量转换为 numpy 数组后的值。还需要注意，self.reset() 函数会清空这些链表

Leaky Integrate-and-Fire 神经元模型，可以看作是带漏电的积分器。其阈下神经动力学方程为：

$$\tau_m \frac{\mathrm{d}V(t)}{\mathrm{d}t} = -(V(t) - V_{reset}) + R_m I(t)$$

- *中文 API*

参数

- **tau** –membrane time constant. tau is shared by all neurons in this layer
- **v_threshold** –threshold voltage of neurons
- **v_reset** –reset voltage of neurons. If not None, voltage of neurons that just fired spikes will be set to v_reset. If None, voltage of neurons that just fired spikes will subtract v_threshold
- **surrogate_function** –surrogate function for replacing gradient of spiking functions during back-propagation
- **detach_reset** –whether detach the computation graph of reset

- **monitor_state** –whether to set a monitor to recode voltage and spikes of neurons. If `True`, `self.monitor` will be a dictionary with key `h` for recording membrane potential after charging, `v` for recording membrane potential after firing and `s` for recording output spikes. And the value of the dictionary is lists. To save memory, the elements in lists are `numpy` array converted from origin data. Besides, `self.reset()` will clear these lists in the dictionary

The Leaky Integrate-and-Fire neuron, which can be seen as a leaky integrator. The subthreshold neural dynamics of it is as followed:

$$\tau_m \frac{\mathrm{d}V(t)}{\mathrm{d}t} = -(V(t) - V_{reset}) + R_m I(t)$$

extra_repr()

neuronal_charge(*dv: torch.Tensor*)

training: bool

spikingjelly.clock_driven.monitor package

Module contents

class spikingjelly.clock_driven.monitor.**Monitor**(*net: torch.nn.modules.module.Module, device: Optional[str] = None, backend: str = 'numpy'*)

基类：`object`

- *API in English*

  参数

  - **net** (*nn.Module*) –要监视的网络
  - **device** (*str, optional*) –监视数据的存储和处理的设备，仅当 backend 为 `'torch'` 时有效。可以为 `'cpu'`, `'cuda'`, `'cuda:0'` 等，默认为 `None`
  - **backend** (*str, optional*) –监视数据的处理后端。可以为 `'torch'`, `'numpy'`，默认为 `'numpy'`

- *中文 API*

  参数

  - **net** (*nn.Module*) –Network to be monitored

- **device** (*str, optional*) –Device carrying and processing monitored data. Only take effect when backend is set to `'torch'`. Can be `'cpu'`, `'cuda'`, `'cuda:0'`, et al., defaults to `None`
- **backend** (*str, optional*) –Backend processing monitored data, can be `'torch'`, `'numpy'`, defaults to `'numpy'`

enable()

- *API in English*

启用 Monitor 的监视功能，开始记录数据

- *中文 API*

Enable Monitor. Start recording data.

disable()

- *API in English*

禁用 Monitor 的监视功能，不再记录数据

- *中文 API*

Disable Monitor. Stop recording data.

forward_hook(*module*, *input*, *output*)

reset()

- *API in English*

清空之前的记录数据

- *中文 API*

Delete previously recorded data

get_avg_firing_rate(*all: bool = True*, *module_name: Optional[str] = None*) → torch.Tensor

- *API in English*

参数

- **all** (*bool, optional*) –是否为所有层的总平均发放率，默认为 `True`
- **module_name** (*str, optional*) –层的名称，仅当 all 为 `False` 时有效

返回 所关心层的平均发放率

返回类型 torch.Tensor or float

- *中文 API*

参数

- **all** (*bool, optional*) –Whether needing firing rate averaged on all layers, defaults to `True`
- **module_name** (*str, optional*) –Name of concerned layer. Only take effect when all is `False`

返回 Averaged firing rate on concerned layers

返回类型 torch.Tensor or float

get_nonfire_ratio(*all: bool = True, module_name: Optional[str] = None*) → torch.Tensor

- *API in English*

参数

- **all** (*bool, optional*) –是否为所有层的静默神经元比例，默认为 `True`
- **module_name** (*str, optional*) –层的名称，仅当 all 为 `False` 时有效

返回 所关心层的静默神经元比例

返回类型 torch.Tensor or float

- *中文 API*

参数

- **all** (*bool, optional*) –Whether needing ratio of silent neurons of all layers, defaults to `True`
- **module_name** (*str, optional*) –Name of concerned layer. Only take effect when all is `False`

返回 Ratio of silent neurons on concerned layers

返回类型 torch.Tensor or float

spikingjelly.clock_driven.optim package

Module contents

spikingjelly.clock_driven.rnn package

Module contents

spikingjelly.clock_driven.rnn.**bidirectional_rnn_cell_forward**(*cell: torch.nn.modules.module.Module, cell_reverse: torch.nn.modules.module.Module, x: torch.Tensor, states: torch.Tensor, states_reverse: torch.Tensor*)

参数

- **cell** (*nn.Module*) –正向 RNN cell，输入是正向序列
- **cell_reverse** (*nn.Module*) –反向的 RNN cell，输入是反向序列
- **x** (*torch.Tensor*) –shape = [T, batch_size, input_size] 的输入
- **states** (*torch.Tensor*) –正向 RNN cell 的起始状态若 RNN cell 只有单个隐藏状态，则 shape = [batch_size, hidden_size]；否则 shape = [states_num, batch_size, hidden_size]
- **states_reverse** –反向 RNN cell 的起始状态若 RNN cell 只有单个隐藏状态，则 shape = [batch_size, hidden_size]；否则 shape = [states_num, batch_size, hidden_size]

返回

y, ss, ss_r

y: torch.Tensor shape = [T, batch_size, 2 * hidden_size] 的输出。y[t] 由正向 cell 在 t 时刻和反向 cell 在 T - t - 1 时刻的输出拼接而来

ss: torch.Tensor shape 与 states 相同，正向 cell 在 T-1 时刻的状态

ss_r: torch.Tensor shape 与 states_reverse 相同，反向 cell 在 0 时刻的状态

计算单个正向和反向 RNN cell 沿着时间维度的循环并输出结果和两个 cell 的最终状态。

class spikingjelly.clock_driven.rnn.**SpikingRNNCellBase**(*input_size: int, hidden_size: int, bias=True*)

基类：torch.nn.modules.module.Module

- *API in English*

Spiking RNN Cell 的基类。

参数

- **input_size** (*int*) –输入 x 的特征数
- **hidden_size** (*int*) –隐藏状态 h 的特征数

- **bias** (*bool*) –若为 `False`, 则内部的隐藏层不会带有偏置项 `b_ih` 和 `b_hh`。默认为 `True`

注解: 所有权重和偏置项都会按照 $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ 进行初始化。其中 $k = \frac{1}{}$

- *中文 API*

The base class of Spiking RNN Cell.

参数

- **input_size** (*int*) –The number of expected features in the input `x`
- **hidden_size** (*int*) –The number of features in the hidden state `h`
- **bias** (*bool*) –If `False`, then the layer does not use bias weights `b_ih` and `b_hh`. Default: `True`

Note

All the weights and biases are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{}$

reset_parameters()

- *API in English*

初始化所有可学习参数。

- *中文 API*

Initialize all learnable parameters.

weight_ih()

- *API in English*

返回 输入到隐藏状态的连接权重

返回类型 torch.Tensor

- *中文 API*

返回 the learnable input-hidden weights

返回类型 torch.Tensor

weight_hh()

- *API in English*

返回 隐藏状态到隐藏状态的连接权重

返回类型 torch.Tensor

- *中文 API*

返回 the learnable hidden-hidden weights

返回类型 torch.Tensor

bias_ih()

- *API in English*

返回 输入到隐藏状态的连接偏置项

返回类型 torch.Tensor

- *中文 API*

返回 the learnable input-hidden bias

返回类型 torch.Tensor

bias_hh()

- *API in English*

返回 隐藏状态到隐藏状态的连接偏置项

返回类型 torch.Tensor

- *中文 API*

返回 the learnable hidden-hidden bias

返回类型 torch.Tensor

training: **bool**

class spikingjelly.clock_driven.rnn.**SpikingRNNBase**(*input_size*, *hidden_size*, *num_layers*, *bias=True*, *dropout_p=0*, *invariant_dropout_mask=False*, *bidirectional=False*, **args*, ***kwargs*)

基类: torch.nn.modules.module.Module

- *API in English*

多层 脉冲 RNN 的基类。

参数

- **input_size** (*int*) –输入 x 的特征数
- **hidden_size** (*int*) –隐藏状态 h 的特征数

- **num_layers** (*int*) –内部 RNN 的层数，例如 `num_layers = 2` 将会创建堆栈式的两层 RNN，第 1 层接收第 0 层的输出作为输入，并计算最终输出
- **bias** (*bool*) –若为 `False`, 则内部的隐藏层不会带有偏置项 `b_ih` 和 `b_hh`。默认为 `True`
- **dropout_p** (*float*) –若非 `0`，则除了最后一层，每个 RNN 层后会增加一个丢弃概率为 `dropout_p` 的 *Dropout* 层。默认为 `0`
- **invariant_dropout_mask** (*bool*) –若为 `False`，则使用普通的 *Dropout*；若为 `True`，则使用 SNN 中特有的，*mask* 不随着时间变化的 *Dropout*‘，参见*Dropout*。默认为 `False`
- **bidirectional** (*bool*) –若为 `True`，则使用双向 RNN。默认为 `False`
- **args** –子类使用的额外参数
- **kwargs** –子类使用的额外参数

- *中文 API*

The base-class of a multi-layer *spiking* RNN.

参数

- **input_size** (*int*) –The number of expected features in the input `x`
- **hidden_size** (*int*) –The number of features in the hidden state `h`
- **num_layers** (*int*) –Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results
- **bias** (*bool*) –If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **dropout_p** (*float*) –If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **invariant_dropout_mask** (*bool*) –If `False`, use the naive *Dropout*; If `True`, use the dropout in SNN that *mask* doesn’t change in different time steps, see `Dropout` for more information. Defaule: `False`
- **bidirectional** (*bool*) –If `True`, becomes a bidirectional LSTM. Default: `False`
- **args** –additional arguments for sub-class
- **kwargs** –additional arguments for sub-class

create_cells(**args*, ***kwargs*)

- *API in English*

参数

- **args** –子类使用的额外参数
- **kwargs** –子类使用的额外参数

返回 若 `self.bidirectional == True` 则会返回正反两个堆栈式 RNN；否则返回单个堆栈式 RNN

返回类型 nn.Sequential

- *中文 API*

参数

- **args** –additional arguments for sub-class
- **kwargs** –additional arguments for sub-class

返回 If `self.bidirectional == True`, return a RNN for forward direction and a RNN for reverse direction; else, return a single stacking RNN

返回类型 nn.Sequential

static base_cell()

- *API in English*

返回 构成该 RNN 的基本 RNN Cell。例如对于*SpikingLSTM*，返回的是*SpikingLSTMCell*

返回类型 nn.Module

- *中文 API*

返回 The base cell of this RNN. E.g., in *SpikingLSTM* this function will return *SpikingLSTMCell*

返回类型 nn.Module

static states_num()

- *API in English*

返回 状态变量的数量。例如对于*SpikingLSTM*，由于其输出是 `h` 和 `c`，因此返回 `2`；而对于*SpikingGRU*，由于其输出是 `h`，因此返回 `1`

返回类型 int

- *中文 API*

返回 The states number. E.g., for `SpikingLSTM` the output are `h` and `c`, this function will return `2`; for `SpikingGRU` the output is `h`, this function will return `1`

返回类型 int

forward(*x: torch.Tensor*, *states=None*)

- *API in English*

参数

- **x** (*torch.Tensor*) – `shape = [T, batch_size, input_size]`，输入序列
- **states** (*torch.Tensor or tuple*) – `self.states_num()` 为 `1` 时是单个 tensor，否则是一个 tuple，包含 `self.states_num()` 个 tensors。所有的 tensor 的尺寸均为 `shape = [num_layers * num_directions, batch, hidden_size]`，包含 `self.states_num()` 个初始状态如果 RNN 是双向的，`num_directions` 为 `2`，否则为 `1`

返回

output, output_states output: torch.Tensor

`shape = [T, batch, num_directions * hidden_size]`，最后一层在所有时刻的输出

output_states: torch.Tensor or tuple `self.states_num()` 为 `1` 时是单个 tensor，否则是一个 tuple，包含 `self.states_num()` 个 tensors。所有的 tensor 的尺寸均为 `shape = [num_layers * num_directions, batch, hidden_size]`，包含 `self.states_num()` 个最后时刻的状态

- *中文 API*

参数

- **x** (*torch.Tensor*) – `shape = [T, batch_size, input_size]`, tensor containing the features of the input sequence
- **states** (*torch.Tensor or tuple*) – a single tensor when `self.states_num()` is `1`, otherwise a tuple with `self.states_num()` tensors. `shape = [num_layers * num_directions, batch, hidden_size]` for all tensors, containing the `self.states_num()` initial states for each element in the batch. If the RNN is bidirectional, `num_directions` should be `2`, else it should be `1`

返回

output, output_states output: torch.Tensor

`shape = [T, batch, num_directions * hidden_size]`, tensor containing the output features from the last layer of the RNN, for each `t`

output_states: torch.Tensor or tuple a single tensor when `self.states_num()` is `1`, otherwise a tuple with `self.states_num()` tensors. `shape = [num_layers * num_directions, batch, hidden_size]` for all tensors, containing the `self.states_num()` states for `t = T - 1`

training: **bool**

class spikingjelly.clock_driven.rnn.**SpikingLSTMCell**(*input_size: int, hidden_size: int, bias=True, surrogate_function1=Erf(alpha=2.0, spiking=True), surrogate_function2=None*)

基类：*spikingjelly.clock_driven.rnn.SpikingRNNCellBase*

- *API in English*

脉冲长短时记忆 (LSTM) cell, 最先由 Long Short-Term Memory Spiking Networks and Their Applications 一文提出。

$$
\begin{aligned}
i &= \Theta(W_{ii}x + b_{ii} + W_{hi}h + b_{hi}) \\
f &= \Theta(W_{if}x + b_{if} + W_{hf}h + b_{hf}) \\
g &= \Theta(W_{ig}x + b_{ig} + W_{hg}h + b_{hg}) \\
o &= \Theta(W_{io}x + b_{io} + W_{ho}h + b_{ho}) \\
c' &= f * c + i * g \\
h' &= o * c'
\end{aligned}
$$

其中 Θ 是 heaviside 阶跃函数（脉冲函数）, and $*$ 是 Hadamard 点积，即逐元素相乘。

参数

- **input_size** (*int*) –输入 `x` 的特征数
- **hidden_size** (*int*) –隐藏状态 `h` 的特征数
- **bias** (*bool*) –若为 `False`, 则内部的隐藏层不会带有偏置项 `b_ih` 和 `b_hh`。默认为 `True`
- **surrogate_function1** (*spikingjelly.clock_driven.surrogate.SurrogateFunctionBase*) –反向传播时用来计算脉冲函数梯度的替代函数, 计算 `i`, `f`, `o` 反向传播时使用
- **surrogate_function2** (*None or spikingjelly.clock_driven.surrogate.SurrogateFunctionBase*) –反向传播时用来计算脉冲函数梯度的替代函数, 计算 `g` 反向传播时使用。若为 `None`, 则设置成 `surrogate_function1`。默认为 `None`

注解: 所有权重和偏置项都会按照 $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ 进行初始化。其中 $k = \frac{1}{}$

示例代码:

```
T = 6
batch_size = 2
input_size = 3
hidden_size = 4
rnn = rnn.SpikingLSTMCell(input_size, hidden_size)
input = torch.randn(T, batch_size, input_size) * 50
h = torch.randn(batch_size, hidden_size)
c = torch.randn(batch_size, hidden_size)

output = []
for t in range(T):
    h, c = rnn(input[t], (h, c))
    output.append(h)
print(output)
```

- *中文 API*

A *spiking* long short-term memory (LSTM) cell, which is firstly proposed in Long Short-Term Memory Spiking Networks and Their Applications.

$$
\begin{aligned}
i &= \Theta(W_{ii}x + b_{ii} + W_{hi}h + b_{hi}) \\
f &= \Theta(W_{if}x + b_{if} + W_{hf}h + b_{hf}) \\
g &= \Theta(W_{ig}x + b_{ig} + W_{hg}h + b_{hg}) \\
o &= \Theta(W_{io}x + b_{io} + W_{ho}h + b_{ho}) \\
c' &= f * c + i * g \\
h' &= o * c'
\end{aligned}
$$

where Θ is the heaviside function, and $*$ is the Hadamard product.

参数

- **input_size** (*int*) –The number of expected features in the input `x`
- **hidden_size** (The number of features in the hidden state `h`) –int
- **bias** (*bool*) –If `False`, then the layer does not use bias weights `b_ih` and `b_hh`. Default: `True`
- **surrogate_function1** (`spikingjelly.clock_driven.surrogate.SurrogateFunctionBase`) –surrogate function for replacing gradient of spiking functions during back-propagation, which is used for generating `i`, `f`, `o`
- **surrogate_function2** (*None or* `spikingjelly.clock_driven.surrogate.SurrogateFunctionBase`) –surrogate function for replacing gradient of spiking functions during back-propagation, which is used for generating `g`. If `None`, the

surrogate function for generating `g` will be set as `surrogate_function1`. Default: `None`

Note

All the weights and biases are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{}$

Examples:

```
T = 6
batch_size = 2
input_size = 3
hidden_size = 4
rnn = rnn.SpikingLSTMCell(input_size, hidden_size)
input = torch.randn(T, batch_size, input_size) * 50
h = torch.randn(batch_size, hidden_size)
c = torch.randn(batch_size, hidden_size)

output = []
for t in range(T):
    h, c = rnn(input[t], (h, c))
    output.append(h)
print(output)
```

forward(*x: torch.Tensor*, *hc=None*)

- *API in English*

参数

- **x** (*torch.Tensor*) –shape = [batch_size, input_size] 的输入
- **hc** (*tuple or None*) –(h_0, c_0) h_0 : torch.Tensor

  shape = [batch_size, hidden_size]，起始隐藏状态

  c_0 [torch.Tensor] shape = [batch_size, hidden_size]，起始细胞状态

  如果不提供 (h_0, c_0)，h_0 默认 c_0 默认为 0

返回

(h_1, c_1) : h_1 : torch.Tensor

shape = [batch_size, hidden_size]，下一个时刻的隐藏状态

c_1 [torch.Tensor] shape = [batch_size, hidden_size]，下一个时刻的细胞状态

返回类型 tuple

- *中文 API*

参数

- **x** (*torch.Tensor*) –the input tensor with `shape = [batch_size, input_size]`
- **hc** (*tuple or None*) –(h_0, c_0) h_0 : torch.Tensor

  `shape = [batch_size, hidden_size]`, tensor containing the initial hidden state for each element in the batch

  c_0 [torch.Tensor] `shape = [batch_size, hidden_size]`, tensor containing the initial cell state for each element in the batch

  If (h_0, c_0) is not provided, both `h_0` and `c_0` default to zero

返回

(h_1, c_1) : h_1 : torch.Tensor

`shape = [batch_size, hidden_size]`, tensor containing the next hidden state for each element in the batch

c_1 [torch.Tensor] `shape = [batch_size, hidden_size]`, tensor containing the next cell state for each element in the batch

返回类型 tuple

training: bool

class spikingjelly.clock_driven.rnn.**SpikingLSTM**(*input_size, hidden_size, num_layers, bias=True, dropout_p=0, invariant_dropout_mask=False, bidirectional=False, surrogate_function1=Erf(alpha=2.0, spiking=True), surrogate_function2=None*)

基类: *spikingjelly.clock_driven.rnn.SpikingRNNBase*

- *API in English*

多层 '脉冲 ' 长短时记忆 LSTM, 最先由 Long Short-Term Memory Spiking Networks and Their Applications 一文提出。

每一层的计算按照

$$
\begin{aligned}
i_t &= \Theta(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \Theta(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \Theta(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \Theta(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t * c_{t-1} + i_t * g_t \\
h_t &= o_t * c'_{t-1}
\end{aligned}
$$

其中 h_t 是 t 时刻的隐藏状态，c_t 是 t 时刻的细胞状态，h_{t-1} 是该层 $t-1$ 时刻的隐藏状态或起始状态，i_t，f_t，g_t，o_t 分别是输入，遗忘，细胞，输出门，Θ 是 heaviside 阶跃函数（脉冲函数），and $*$ 是 Hadamard 点积，即逐元素相乘。

参数

- **input_size** (*int*) –输入 x 的特征数
- **hidden_size** (*int*) –隐藏状态 h 的特征数
- **num_layers** (*int*) –内部 RNN 的层数，例如 `num_layers = 2` 将会创建堆栈式的两层 RNN，第 1 层接收第 0 层的输出作为输入，并计算最终输出
- **bias** (*bool*) –若为 `False`, 则内部的隐藏层不会带有偏置项 `b_ih` 和 `b_hh`。默认为 `True`
- **dropout_p** (*float*) –若非 `0`，则除了最后一层，每个 RNN 层后会增加一个丢弃概率为 `dropout_p` 的 *Dropout* 层。默认为 `0`
- **invariant_dropout_mask** (*bool*) –若为 `False`，则使用普通的 *Dropout*；若为 `True`，则使用 SNN 中特有的，*mask* 不随着时间变化的 *Dropout*‘，参见`Dropout`。默认为 `False`
- **bidirectional** (*bool*) –若为 `True`，则使用双向 RNN。默认为 `False`
- **surrogate_function1** (`spikingjelly.clock_driven.surrogate.SurrogateFunctionBase`) –反向传播时用来计算脉冲函数梯度的替代函数, 计算 `i`, `f`, `o` 反向传播时使用
- **surrogate_function2** (*None or* `spikingjelly.clock_driven.surrogate.SurrogateFunctionBase`) –反向传播时用来计算脉冲函数梯度的替代函数, 计算 `g` 反向传播时使用。若为 `None`, 则设置成 `surrogate_function1`。默认为 `None`

- *中文 API*

The *spiking* multi-layer long short-term memory (LSTM), which is firstly proposed in Long Short-Term Memory Spiking Networks and Their Applications.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \Theta(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \Theta(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \Theta(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \Theta(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t * c_{t-1} + i_t * g_t \\
h_t &= o_t * c'_{t-1}
\end{aligned}
$$

where h_t is the hidden state at time *t*, c_t is the cell state at time *t*, x_t is the input at time *t*, h_{t-1} is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. Θ is the heaviside function, and $*$ is the Hadamard product.

参数

- **input_size** (*int*) –The number of expected features in the input `x`
- **hidden_size** (*int*) –The number of features in the hidden state `h`
- **num_layers** (*int*) –Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results
- **bias** (*bool*) –If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **dropout_p** (*float*) –If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **invariant_dropout_mask** (*bool*) –If `False`, use the naive *Dropout*; If `True`, use the dropout in SNN that *mask* doesn't change in different time steps, see `Dropout` for more information. Defaule: `False`
- **bidirectional** (*bool*) –If `True`, becomes a bidirectional LSTM. Default: `False`
- **surrogate_function1** (`spikingjelly.clock_driven.surrogate.SurrogateFunctionBase`) –surrogate function for replacing gradient of spiking functions during back-propagation, which is used for generating `i`, `f`, `o`
- **surrogate_function2** (*None or* `spikingjelly.clock_driven.surrogate.SurrogateFunctionBase`) –surrogate function for replacing gradient of spiking functions during back-propagation, which is used for generating `g`. If `None`, the surrogate function for generating `g` will be set as `surrogate_function1`. Default: `None`

static base_cell()

static states_num()

training: bool

class spikingjelly.clock_driven.rnn.**SpikingVanillaRNNCell**(*input_size: int, hidden_size: int, bias=True, surrogate_function=Erf(alpha=2.0, spiking=True)*)

基类：spikingjelly.clock_driven.rnn.SpikingRNNCellBase

forward(*x: torch.Tensor, h=None*)

training: bool

class spikingjelly.clock_driven.rnn.**SpikingVanillaRNN**(*input_size, hidden_size, num_layers, bias=True, dropout_p=0, invariant_dropout_mask=False, bidirectional=False, surrogate_function=Erf(alpha=2.0, spiking=True)*)

基类：spikingjelly.clock_driven.rnn.SpikingRNNBase

static base_cell()

static states_num()

training: bool

class spikingjelly.clock_driven.rnn.**SpikingGRUCell**(*input_size: int, hidden_size: int, bias=True, surrogate_function1=Erf(alpha=2.0, spiking=True), surrogate_function2=None*)

基类：spikingjelly.clock_driven.rnn.SpikingRNNCellBase

forward(*x: torch.Tensor, hc=None*)

training: bool

class spikingjelly.clock_driven.rnn.**SpikingGRU**(*input_size, hidden_size, num_layers, bias=True, dropout_p=0, invariant_dropout_mask=False, bidirectional=False, surrogate_function1=Erf(alpha=2.0, spiking=True), surrogate_function2=None*)

基类：spikingjelly.clock_driven.rnn.SpikingRNNBase

static base_cell()

static states_num()

training: bool

spikingjelly.clock_driven.surrogate package

Module contents

spikingjelly.clock_driven.surrogate.**heaviside**(*x: torch.Tensor*)

- *API in English*

参数 **x** –输入 tensor

返回 输出 tensor

heaviside 阶跃函数，定义为

$$g(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

阅读 HeavisideStepFunction 以获得更多信息。

- *中文 API*

参数 **x** –the input tensor

返回 the output tensor

The heaviside function, which is defined by

$$g(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

For more information, see HeavisideStepFunction.

spikingjelly.clock_driven.surrogate.**check_manual_grad**(*primitive_function*, *spiking_function*, *eps=1e-05*)

参数

- **primitive_function** (*callable*) –梯度替代函数的原函数
- **spiking_function** (*callable*) –梯度替代函数
- **eps** (*float*) –最大误差

梯度替代函数的反向传播一般是手写的，可以用此函数去检查手写梯度是否正确。

此函数检查梯度替代函数 spiking_function 的反向传播，与原函数 primitive_function 的反向传播结果是否一致。“一致”被定义为，两者的误差不超过 eps。

示例代码：

```
surrogate.check_manual_grad(surrogate.ATan.primitive_function, surrogate.atan.
↪apply)
```

class spikingjelly.clock_driven.surrogate.**SurrogateFunctionBase**(*alpha*, *spiking=True*)

基类：torch.nn.modules.module.Module

set_spiking_mode(*spiking: bool*)

extra_repr()

static spiking_function(*x*, *alpha*)

static primitive_function(*x*, *alpha*)

forward(*x: torch.Tensor*)

training: bool

class spikingjelly.clock_driven.surrogate.**MultiArgsSurrogateFunctionBase**(*spiking=True*, ***kwargs*)

基类：torch.nn.modules.module.Module

set_spiking_mode(*spiking: bool*)

extra_repr()

training: bool

class spikingjelly.clock_driven.surrogate.**piecewise_quadratic**

基类：torch.autograd.function.Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**PiecewiseQuadratic**(*alpha=1.0*, *spiking=True*)

基类：*spikingjelly.clock_driven.surrogate.SurrogateFunctionBase*

- *API in English*

参数

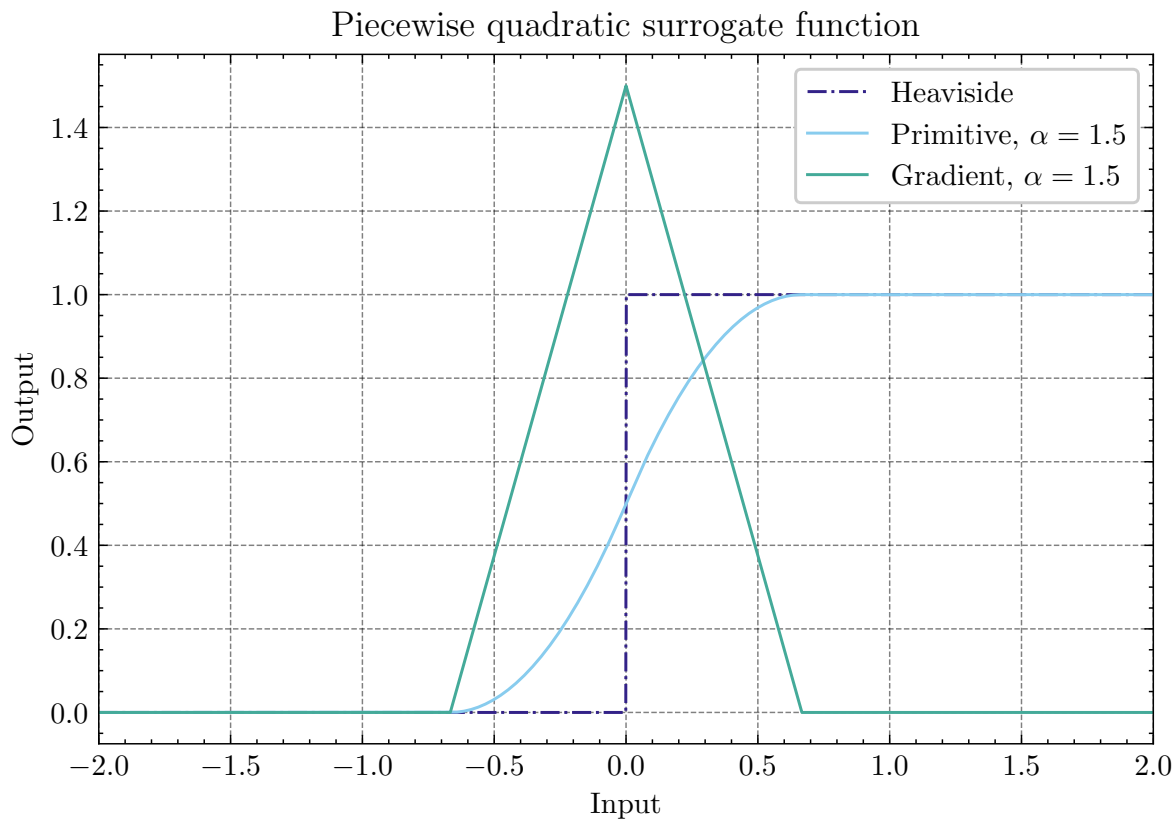
- **alpha** –控制反向传播时梯度的平滑程度的参数
- **spiking** –是否输出脉冲，默认为 True，在前向传播时使用 heaviside 而在反向传播使用替代梯度。若为 False 则不使用替代梯度，前向传播时，使用反向传播时的梯度替代函数对应的原函数

反向传播时使用分段二次函数的梯度（三角形函数）的脉冲发放函数。反向传播为

$$g'(x) = \begin{cases} 0, & |x| > \frac{1}{\alpha} \\ -\alpha^2|x| + \alpha, & |x| \leq \frac{1}{\alpha} \end{cases}$$

对应的原函数为

$$g(x) = \begin{cases} 0, & x < -\frac{1}{\alpha} \\ -\frac{1}{2}\alpha^2|x|x + \alpha x + \frac{1}{2}, & |x| \le \frac{1}{\alpha} \\ 1, & x > \frac{1}{\alpha} \end{cases}$$



该函数在文章[2][4][7][11][13] 中使用。

- *中文 API*

参数

- **alpha** –parameter to control smoothness of gradient
- **spiking** –whether output spikes. The default is `True` which means that using `heaviside`

[2] Esser S K, Merolla P A, Arthur J V, et al. Convolutional networks for fast, energy-efficient neuromorphic computing[J]. Proceedings of the national academy of sciences, 2016, 113(41): 11441-11446.

[4] Wu Y, Deng L, Li G, et al. Spatio-temporal backpropagation for training high-performance spiking neural networks[J]. Frontiers in neuroscience, 2018, 12: 331.

[7] Bellec G, Salaj D, Subramoney A, et al. Long short-term memory and learning-to-learn in networks of spiking neurons[C]//Proceedings of the 32nd International Conference on Neural Information Processing Systems. 2018: 795-805.

[11] Neftci E O, Mostafa H, Zenke F. Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks[J]. IEEE Signal Processing Magazine, 2019, 36(6): 51-63.

[13] Panda P, Aketi S A, Roy K. Toward scalable, efficient, and accurate deep spiking neural networks with backward residual connections, stochastic softmax, and hybridization[J]. Frontiers in Neuroscience, 2020, 14.

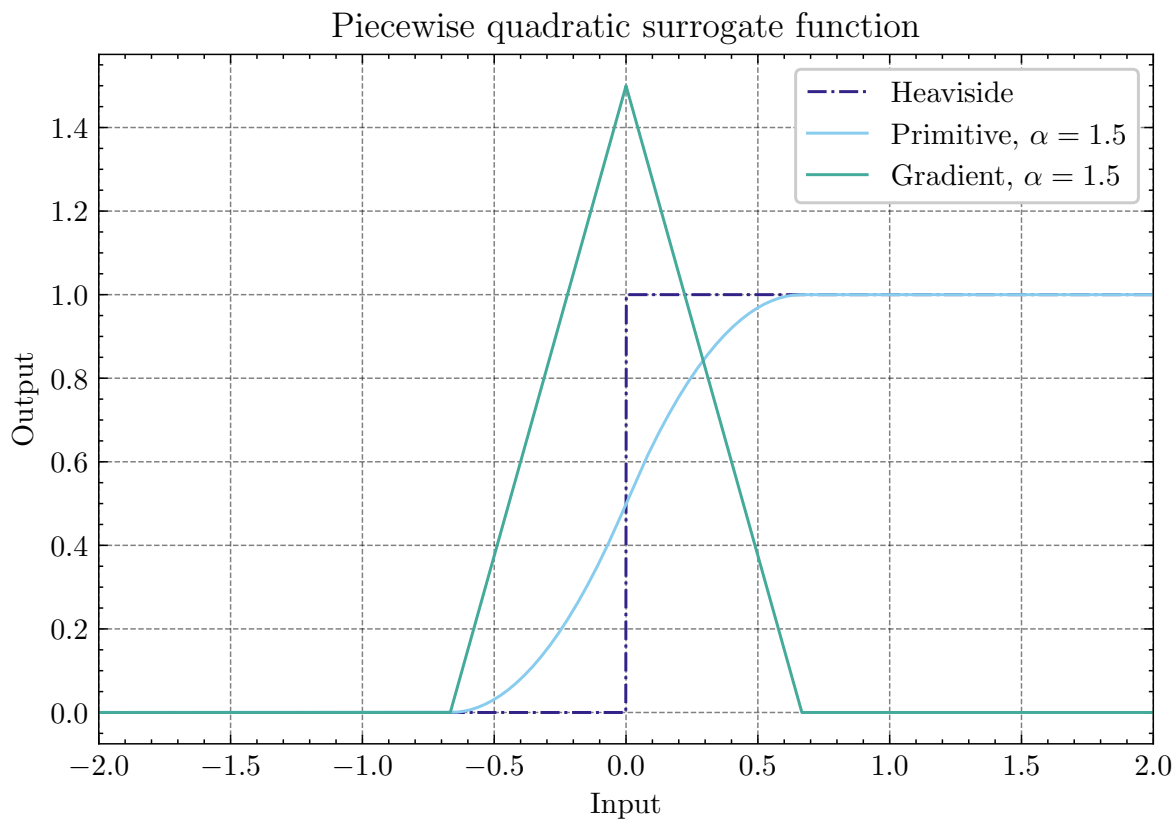
in forward propagation and using surrogate gradient in backward propagation. If `False`, in forward propagation, using the primitive function of the surrogate gradient function used in backward propagation

The piecewise quadratic surrogate spiking function. The gradient is defined by

$$g'(x) = \begin{cases} 0, & |x| > \frac{1}{\alpha} \\ -\alpha^2|x| + \alpha, & |x| \leq \frac{1}{\alpha} \end{cases}$$

The primitive function is defined by

$$g(x) = \begin{cases} 0, & x < -\frac{1}{\alpha} \\ -\frac{1}{2}\alpha^2|x|x + \alpha x + \frac{1}{2}, & |x| \leq \frac{1}{\alpha} \\ 1, & x > \frac{1}{\alpha} \end{cases}$$



The function is used in[?????].

static spiking_function(*x*, *alpha*)

static primitive_function(*x: torch.Tensor*, *alpha*)

training: bool

class spikingjelly.clock_driven.surrogate.**piecewise_exp**

基类: torch.autograd.function.Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**PiecewiseExp**(*alpha=1.0*, *spiking=True*)

基类：*spikingjelly.clock_driven.surrogate.SurrogateFunctionBase*

- *API in English*

参数

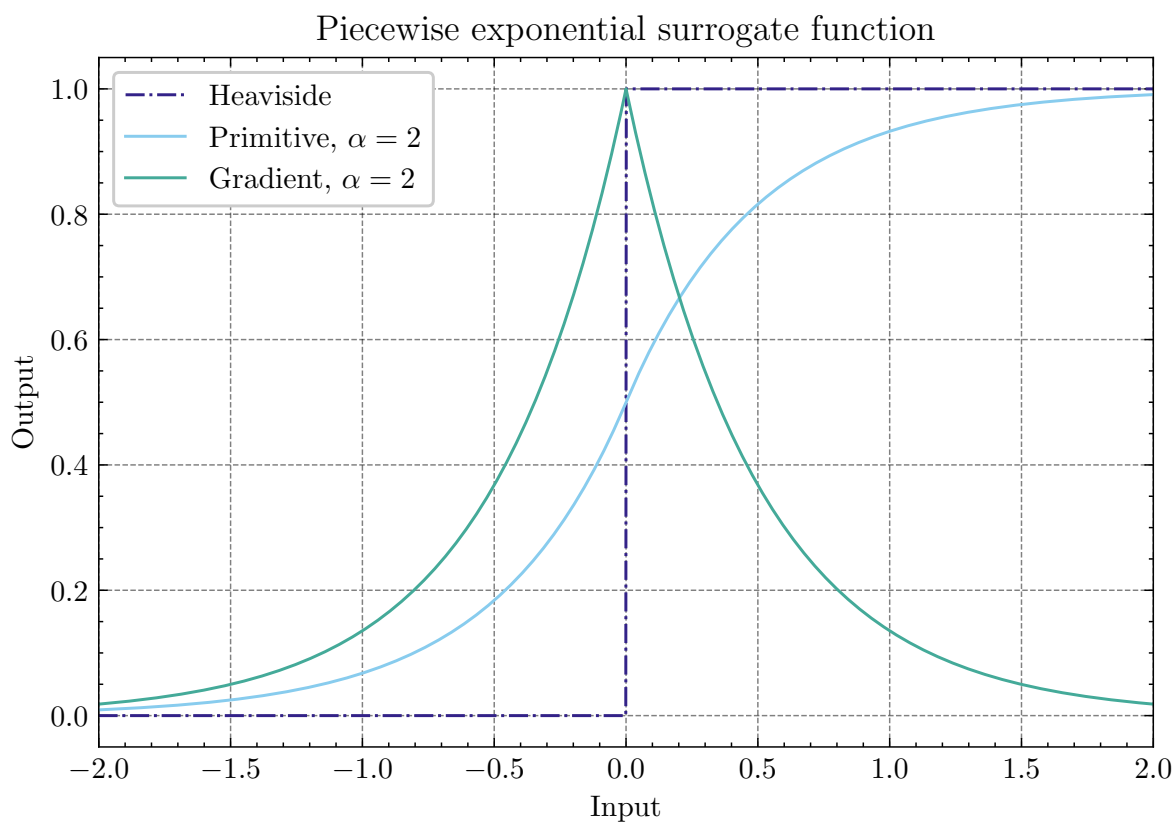
- **alpha** –控制反向传播时梯度的平滑程度的参数
- **spiking** –是否输出脉冲，默认为 `True`，在前向传播时使用 `heaviside` 而在反向传播使用替代梯度。若为 `False` 则不使用替代梯度，前向传播时，使用反向传播时的梯度替代函数对应的原函数

反向传播时使用分段指数函数的梯度的脉冲发放函数。反向传播为

$$g'(x) = \frac{\alpha}{2} e^{-\alpha|x|}$$

对应的原函数为

$$g(x) = \begin{cases} \frac{1}{2}e^{\alpha x}, & x < 0 \\ 1 - \frac{1}{2}e^{-\alpha x}, & x \geq 0 \end{cases}$$



该函数在文章[6?] 中使用。

- *中文 API*

参数

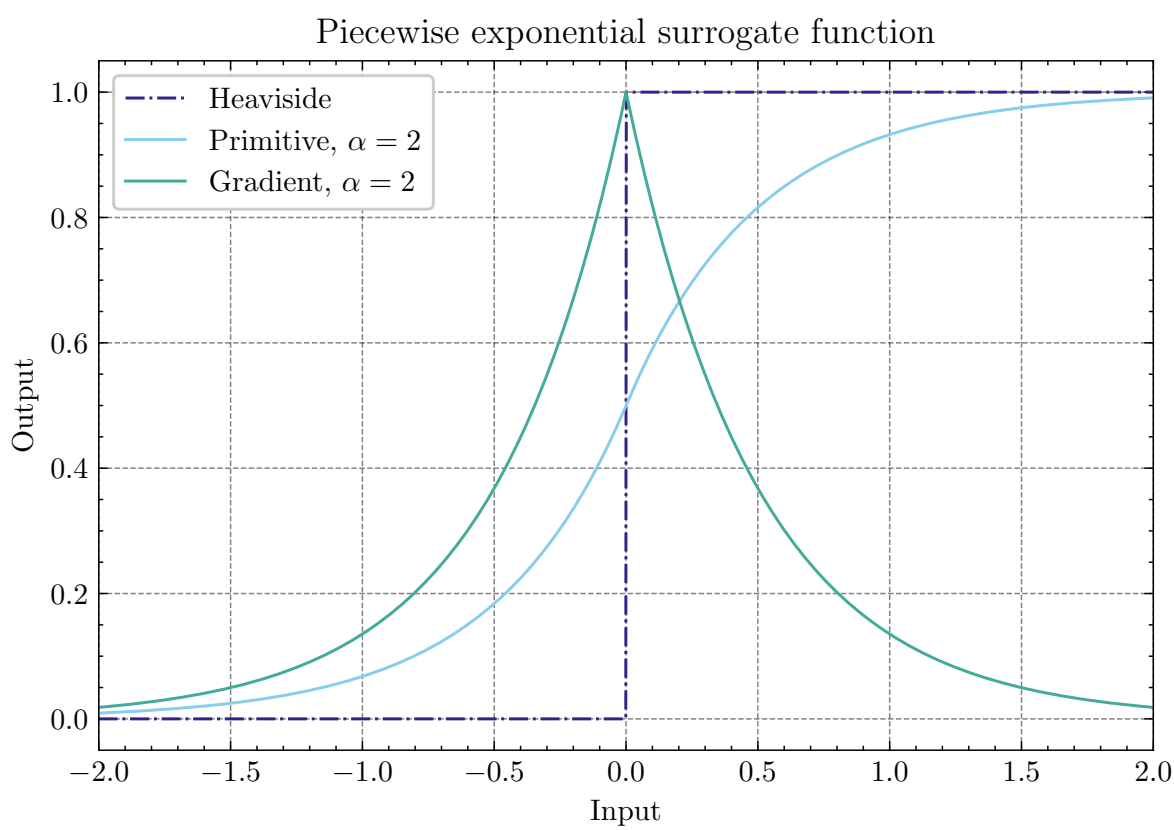
- **alpha** –parameter to control smoothness of gradient
- **spiking** –whether output spikes. The default is `True` which means that using `heaviside` in forward propagation and using surrogate gradient in backward propagation. If `False`, in forward propagation, using the primitive function of the surrogate gradient function used in backward propagation

The piecewise exponential surrogate spiking function. The gradient is defined by

$$g'(x) = \frac{\alpha}{2} e^{-\alpha |x|}$$

The primitive function is defined by

$$g(x) = \begin{cases} \frac{1}{2} e^{\alpha x}, & x < 0 \\ 1 - \frac{1}{2} e^{-\alpha x}, & x \geq 0 \end{cases}$$



The function is used in[??] .

[6] Shrestha S B, Orchard G. SLAYER: spike layer error reassignment in time[C]//Proceedings of the 32nd International Conference on Neural Information Processing Systems. 2018: 1419-1428.

static spiking_function(*x*, *alpha*)

static primitive_function(*x: torch.Tensor*, *alpha*)

training: bool

class spikingjelly.clock_driven.surrogate.**sigmoid**

基类：torch.autograd.function.Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**Sigmoid**(*alpha=1.0*, *spiking=True*)

基类：*spikingjelly.clock_driven.surrogate.SurrogateFunctionBase*

- *API in English*

参数

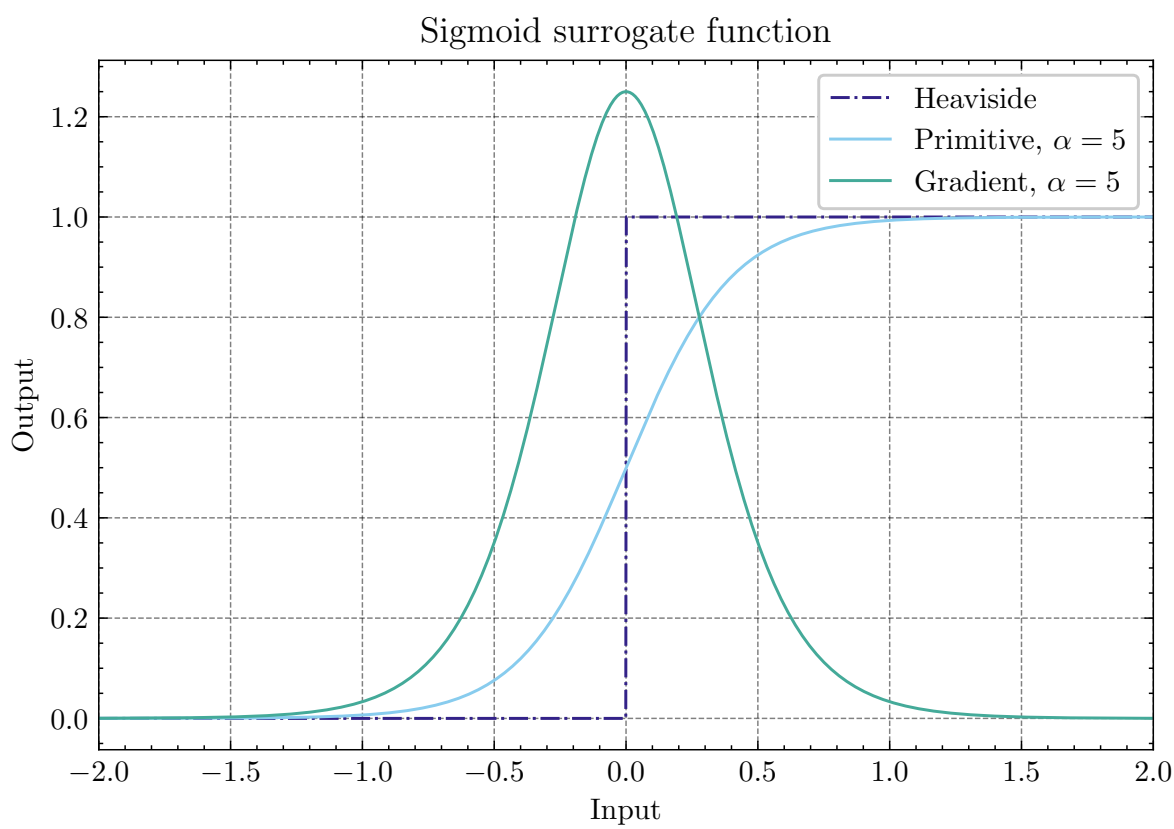
- **alpha** –控制反向传播时梯度的平滑程度的参数
- **spiking** –是否输出脉冲，默认为 True，在前向传播时使用 heaviside 而在反向传播使用替代梯度。若为 False 则不使用替代梯度，前向传播时，使用反向传播时的梯度替代函数对应的原函数

反向传播时使用 sigmoid 的梯度的脉冲发放函数。反向传播为

$$g'(x) = \alpha * (1 - \mathrm{sigmoid}(\alpha x))\mathrm{sigmoid}(\alpha x)$$

对应的原函数为

$$g(x) = \mathrm{sigmoid}(\alpha x) = \frac{1}{1 + e^{-\alpha x}}$$



该函数在文章?[12][14][15] 中使用。

- *中文 API*

参数

- **alpha** –parameter to control smoothness of gradient
- **spiking** –whether output spikes. The default is `True` which means that using `heaviside` in forward propagation and using surrogate gradient in backward propagation. If `False`, in forward propagation, using the primitive function of the surrogate gradient function used in backward propagation

The sigmoid surrogate spiking function. The gradient is defined by

$$g'(x) = \alpha * (1 - \mathrm{sigmoid}(\alpha x))\mathrm{sigmoid}(\alpha x)$$

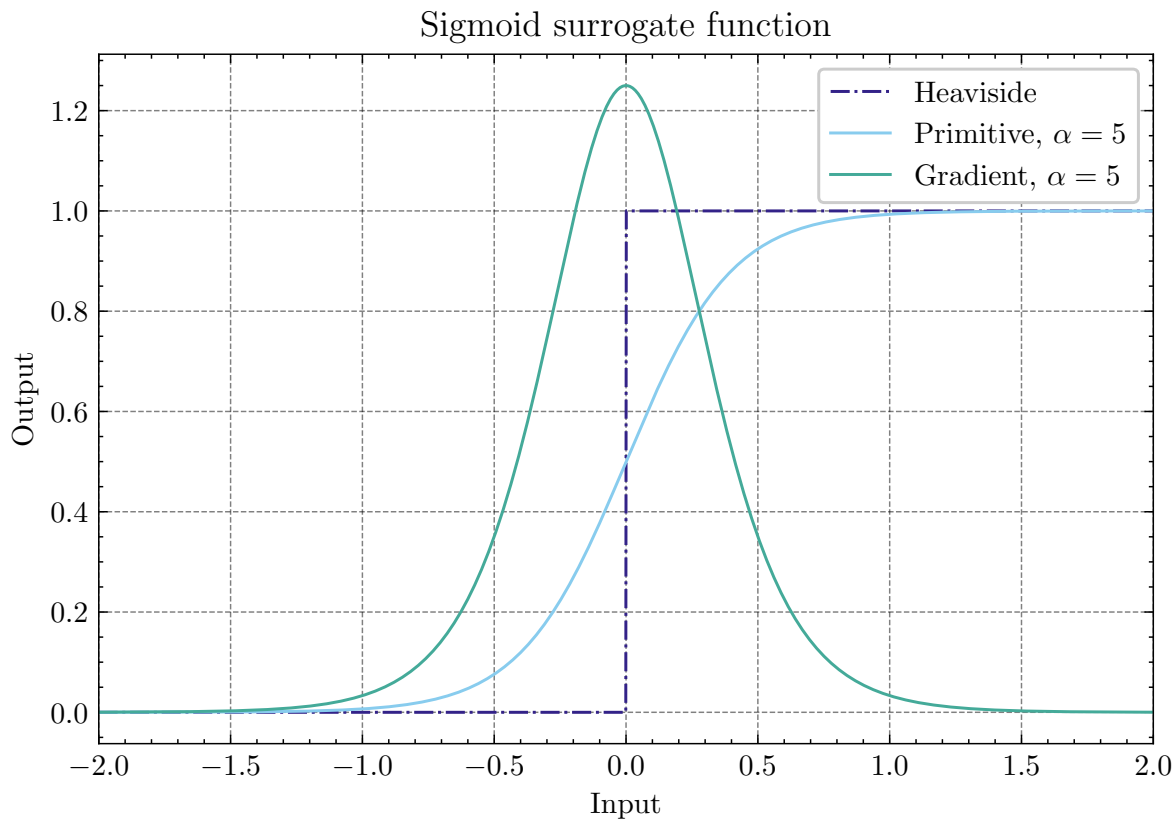
[12] Roy D, Chakraborty I, Roy K. Scaling deep spiking neural networks with binary stochastic activations[C]//2019 IEEE International Conference on Cognitive Computing (ICCC). IEEE, 2019: 50-58.

[14] Lotfi Rezaabad A, Vishwanath S. Long Short-Term Memory Spiking Networks and Their Applications[C]//International Conference on Neuromorphic Systems 2020. 2020: 1-9.

[15] Woźniak S, Pantazi A, Bohnstingl T, et al. Deep learning incorporating biologically inspired neural dynamics and in-memory computing[J]. Nature Machine Intelligence, 2020, 2(6): 325-336.

The primitive function is defined by

$$g(x) = \text{sigmoid}(\alpha x) = \frac{1}{1 + e^{-\alpha x}}$$



The function is used in[????] .

static spiking_function(*x*, *alpha*)

static primitive_function(*x: torch.Tensor*, *alpha*)

training: bool

class spikingjelly.clock_driven.surrogate.**soft_sign**

基类: torch.autograd.function.Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**SoftSign**(*alpha=2.0*, *spiking=True*)

基类: *spikingjelly.clock_driven.surrogate.SurrogateFunctionBase*

- *API in English*

参数

- **alpha** –控制反向传播时梯度的平滑程度的参数

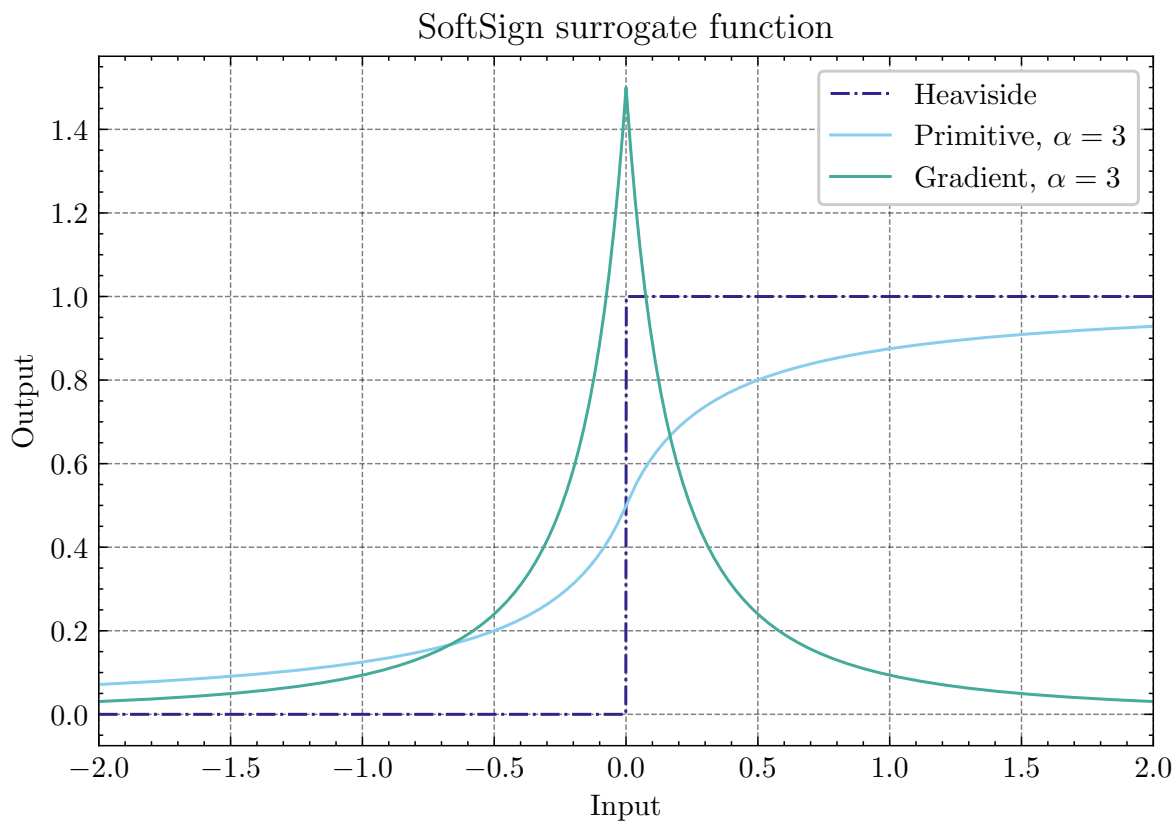
- **spiking** –是否输出脉冲，默认为 `True`，在前向传播时使用 `heaviside` 而在反向传播使用替代梯度。若为 `False` 则不使用替代梯度，前向传播时，使用反向传播时的梯度替代函数对应的原函数

反向传播时使用 soft sign 的梯度的脉冲发放函数。反向传播为

$$g'(x) = \frac{\alpha}{2(1+|\alpha x|)^2} = \frac{1}{2\alpha(\frac{1}{\alpha}+|x|)^2}$$

对应的原函数为

$$g(x) = \frac{1}{2}(\frac{\alpha x}{1+|\alpha x|}+1) = \frac{1}{2}(\frac{x}{\frac{1}{\alpha}+|x|}+1)$$



该函数在文章[8?] 中使用。

- *中文 API*

参数

- **alpha** –parameter to control smoothness of gradient
- **spiking** –whether output spikes. The default is `True` which means that using `heaviside` in forward propagation and using surrogate gradient in backward propagation. If `False`, in forward propagation, using the primitive function of the surrogate gradient function used in backward propagation

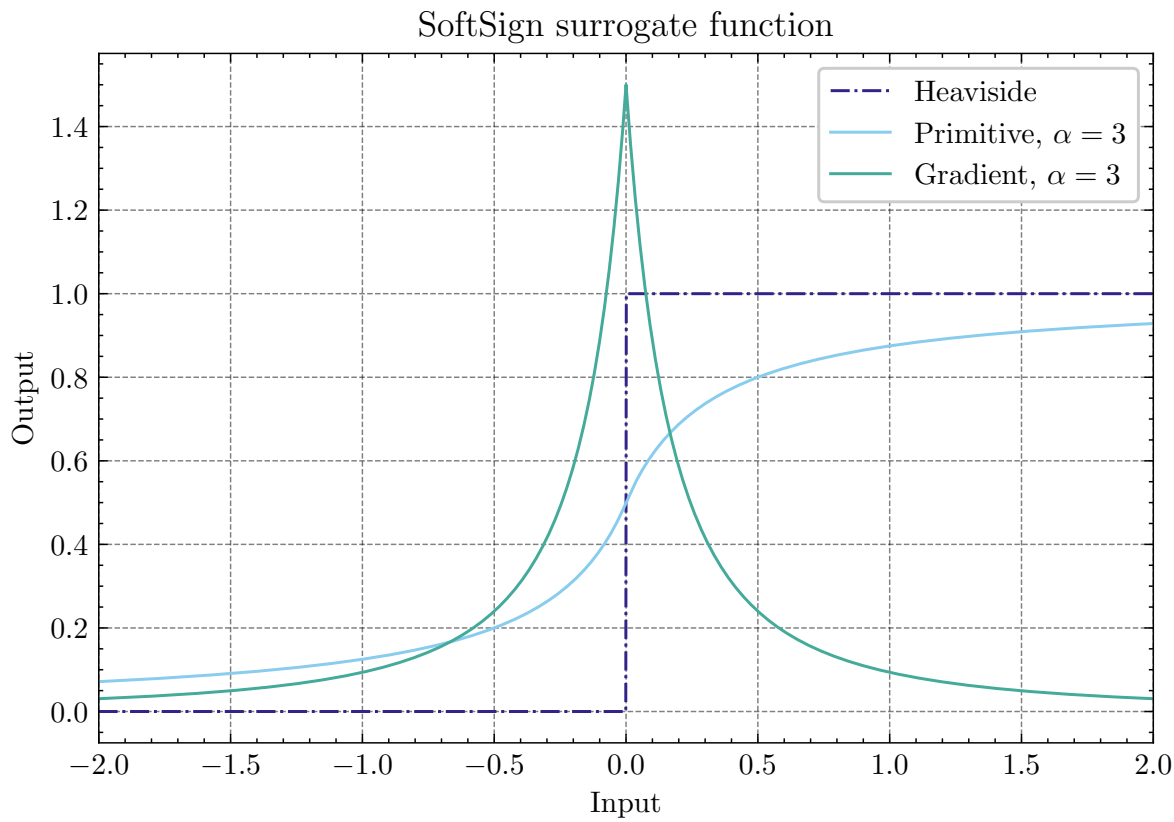
[8] Zenke F, Ganguli S. Superspike: Supervised learning in multilayer spiking neural networks[J]. Neural computation, 2018, 30(6): 1514-1541.

The soft sign surrogate spiking function. The gradient is defined by

$$g'(x) = \frac{\alpha}{2(1+|\alpha x|)^2}$$

The primitive function is defined by

$$g(x) = \frac{1}{2}(\frac{\alpha x}{1+|\alpha x|} + 1)$$



The function is used in[??] .

static spiking_function(*x*, *alpha*)

static primitive_function(*x: torch.Tensor*, *alpha*)

training: bool

class spikingjelly.clock_driven.surrogate.**atan**

基类: torch.autograd.function.Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**ATan**(*alpha=2.0*, *spiking=True*)

基类: *spikingjelly.clock_driven.surrogate.SurrogateFunctionBase*

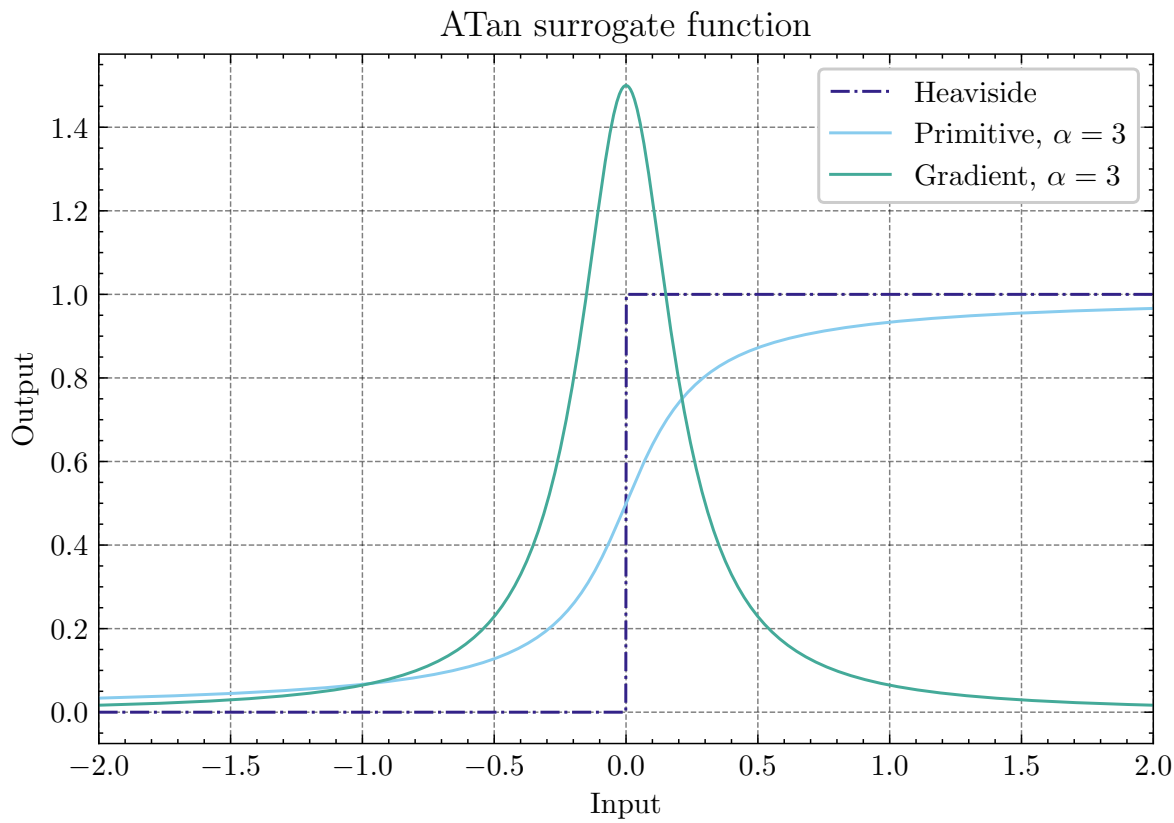
- *API in English*

反向传播时使用反正切函数 arc tangent 的梯度的脉冲发放函数。反向传播为

$$g'(x) = \frac{\alpha}{2(1 + (\frac{\pi}{2}\alpha x)^2)}$$

对应的原函数为

$$g(x) = \frac{1}{\pi}\arctan(\frac{\pi}{2}\alpha x) + \frac{1}{2}$$



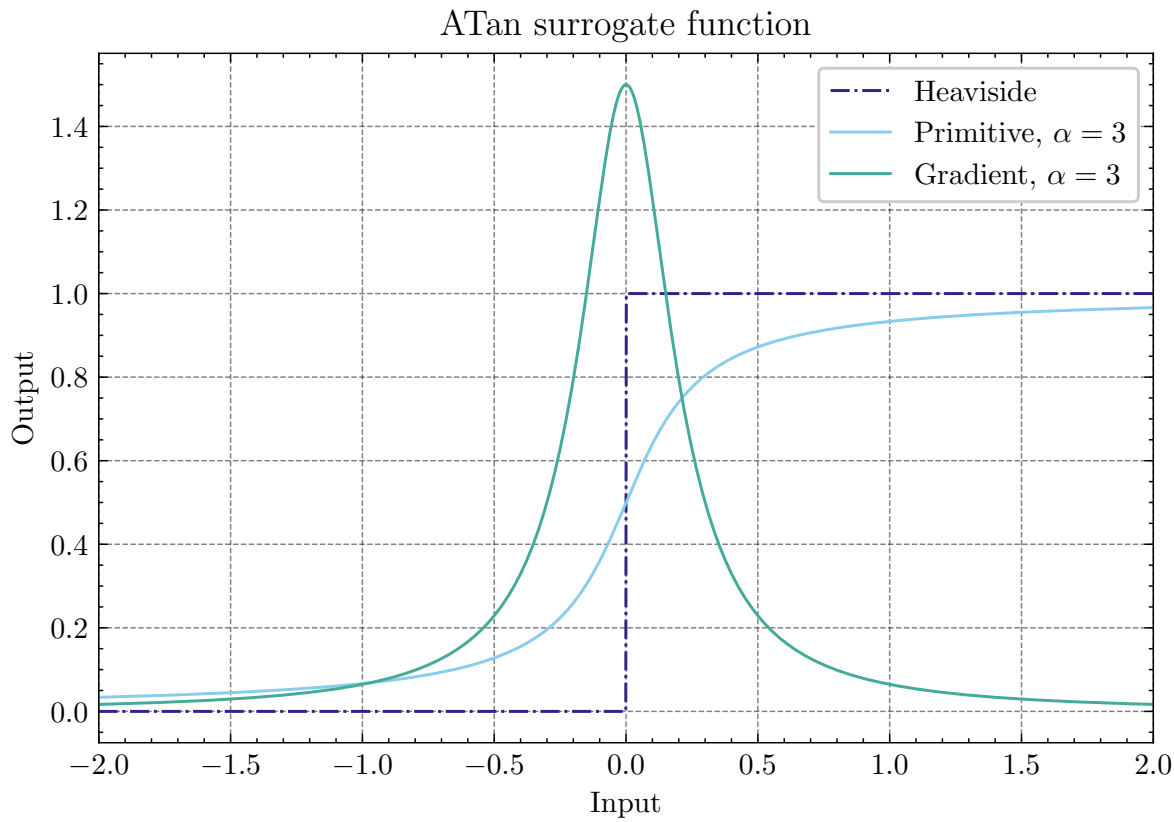
- *中文 API*

The arc tangent surrogate spiking function. The gradient is defined by

$$g'(x) = \frac{\alpha}{2(1 + (\frac{\pi}{2}\alpha x)^2)}$$

The primitive function is defined by

$$g(x) = \frac{1}{\pi}\arctan(\frac{\pi}{2}\alpha x) + \frac{1}{2}$$



static spiking_function(*x*, *alpha*)

static primitive_function(*x: torch.Tensor*, *alpha*)

training: bool

class spikingjelly.clock_driven.surrogate.**nonzero_sign_log_abs**

基类：torch.autograd.function.Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**NonzeroSignLogAbs**(*alpha=1.0*, *spiking=True*)

基类：*spikingjelly.clock_driven.surrogate.SurrogateFunctionBase*

- *API in English*

参数

- **alpha** –控制反向传播时梯度的平滑程度的参数
- **spiking** –是否输出脉冲，默认为 True，在前向传播时使用 heaviside 而在反向传播使用替代梯度。若为 False 则不使用替代梯度，前向传播时，使用反向传播时的梯度替代函数对应的原函数

> **警告：** 原函数的输出范围并不是 (0, 1)。它的优势是反向传播的计算量特别小。

反向传播时使用 NonzeroSignLogAbs 的梯度的脉冲发放函数。反向传播为

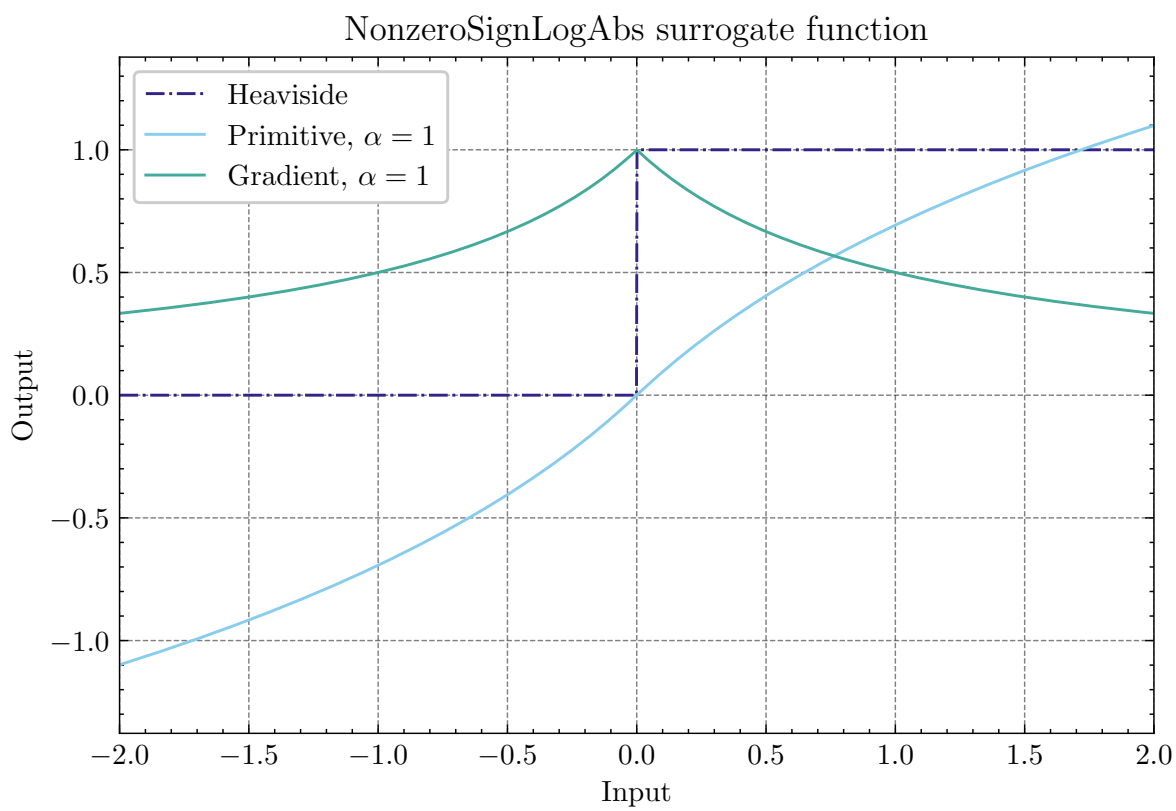
$$g'(x) = \frac{\alpha}{1 + |\alpha x|} = \frac{1}{\frac{1}{\alpha} + |x|}$$

对应的原函数为

$$g(x) = \mathrm{NonzeroSign}(x) \log(|\alpha x| + 1)$$

其中

$$\mathrm{NonzeroSign}(x) = \begin{cases} 1, & x \geq 0 \\ -1, & x < 0 \end{cases}$$



该函数在文章中使用。

- *中文 API*

参数

- **alpha** –parameter to control smoothness of gradient

- **spiking** –whether output spikes. The default is `True` which means that using `heaviside` in forward propagation and using surrogate gradient in backward propagation. If `False`, in forward propagation, using the primitive function of the surrogate gradient function used in backward propagation

---

Warning

The output range the primitive function is not (0, 1). The advantage of this function is that computation cost is small when backward.

The NonzeroSignLogAbs surrogate spiking function. The gradient is defined by

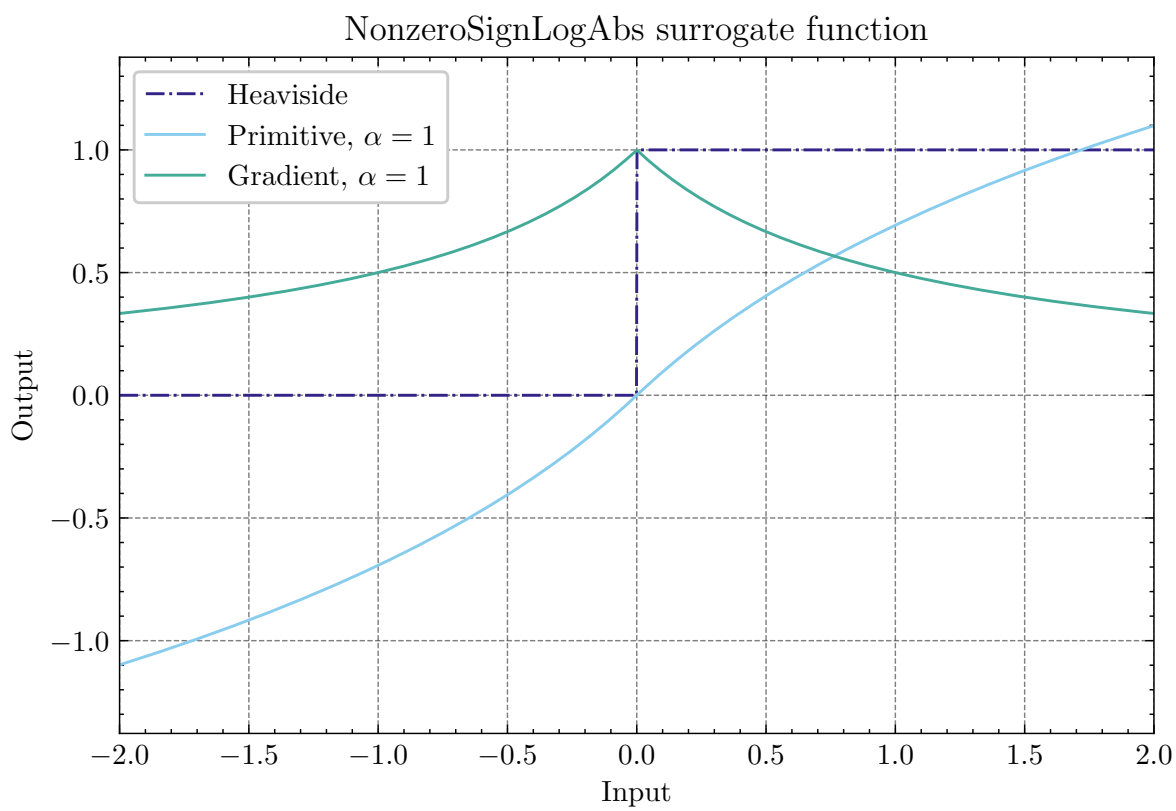
$$g'(x) = \frac{\alpha}{1 + |\alpha x|} = \frac{1}{\frac{1}{\alpha} + |x|}$$

The primitive function is defined by

$$g(x) = \mathrm{NonzeroSign}(x) \log(|\alpha x| + 1)$$

where

$$\mathrm{NonzeroSign}(x) = \begin{cases} 1, & x \geq 0 \\ -1, & x < 0 \end{cases}$$



The function is used in .

static spiking_function(*x*, *alpha*)

static primitive_function(*x: torch.Tensor*, *alpha*)

training: bool

class spikingjelly.clock_driven.surrogate.**erf**

基类：torch.autograd.function.Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**Erf**(*alpha=2.0*, *spiking=True*)

基类：*spikingjelly.clock_driven.surrogate.SurrogateFunctionBase*

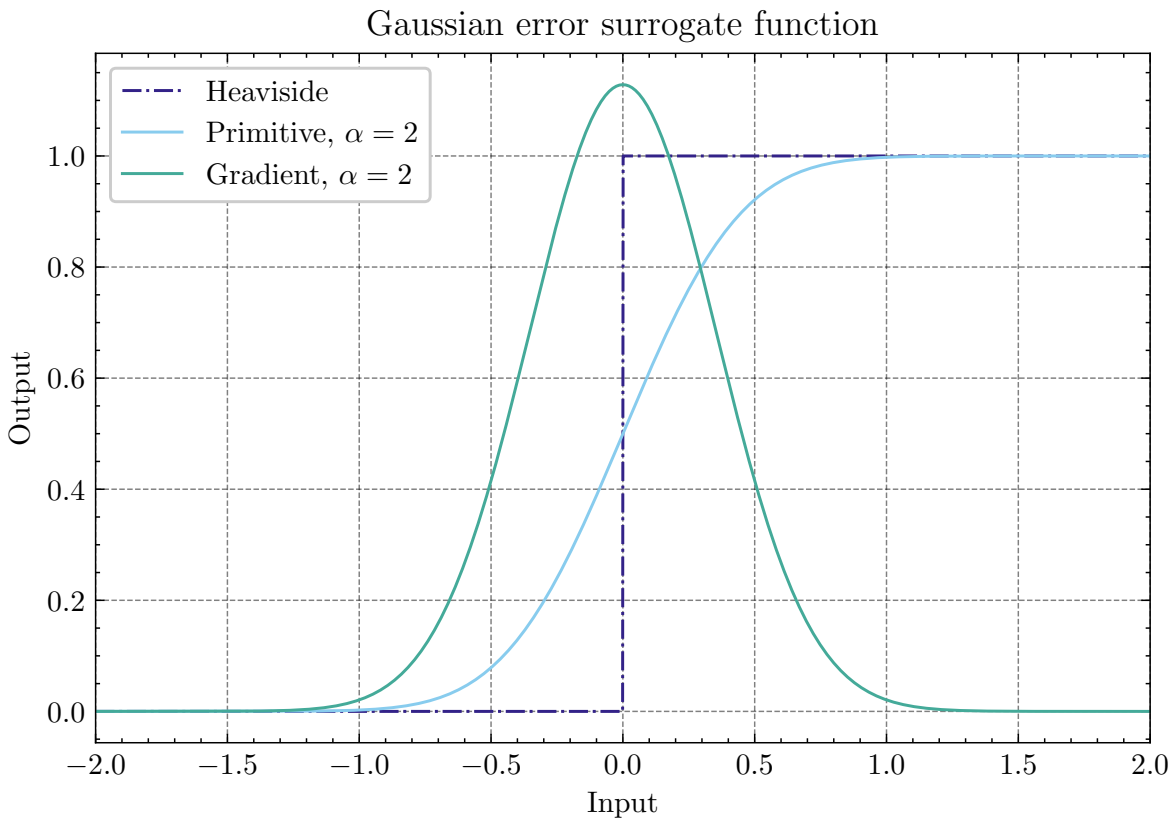
- *API in English*

参数

- **alpha** –控制反向传播时梯度的平滑程度的参数
- **spiking** –是否输出脉冲，默认为 True，在前向传播时使用 heaviside 而在反向传播使用替代梯度。若为 False 则不使用替代梯度，前向传播时，使用反向传播时的梯度替代函数对应的原函数

反向传播时使用高斯误差函数 (erf) 的梯度的脉冲发放函数。反向传播为

$$g'(x) = \frac{\alpha}{\sqrt{\pi}} e^{-\alpha^2 x^2}$$



对应的原函数为

该函数在文章[1?18] 中使用。

- *中文 API*

参数

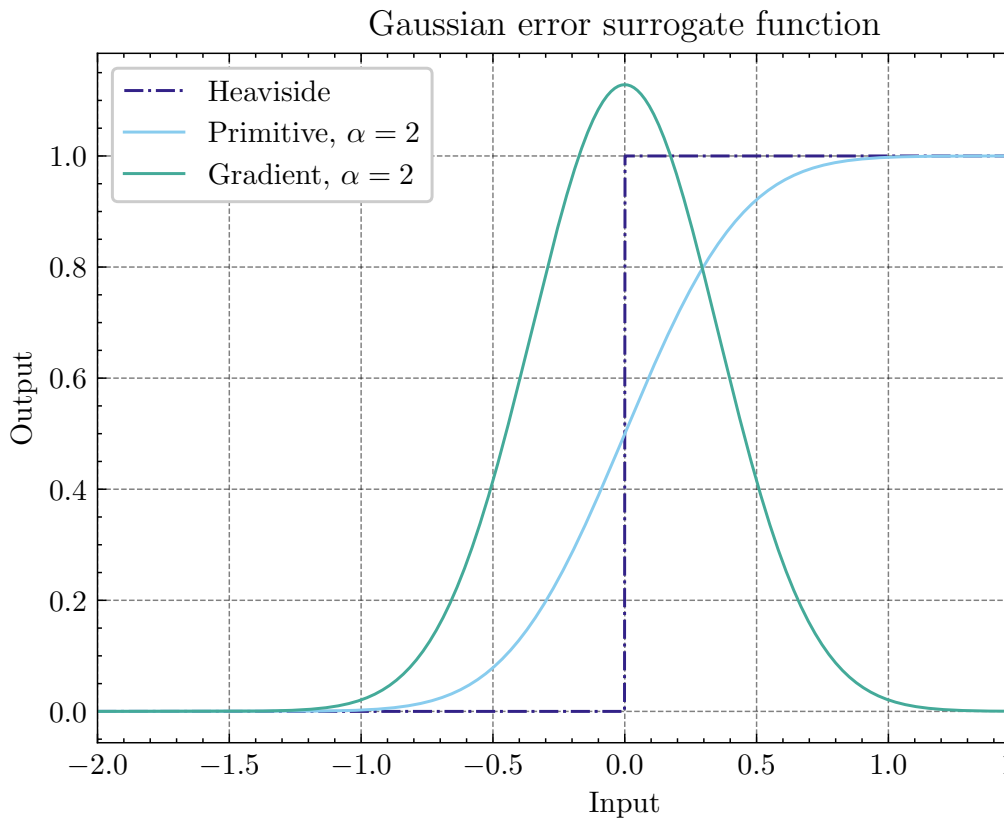
- **alpha** –parameter to control smoothness of gradient
- **spiking** –whether output spikes. The default is `True` which means that using `heaviside` in forward propagation and using surrogate gradient in backward propagation. If `False`, in forward propagation, using the primitive function of the surrogate gradient function used in backward propagation

The Gaussian error (erf) surrogate spiking function. The gradient is defined by

$$g'(x) = \frac{\alpha}{\sqrt{\pi}} e^{-\alpha^2 x^2}$$

[1] Esser S K, Appuswamy R, Merolla P, et al. Backpropagation for energy-efficient neuromorphic computing[J]. Advances in neural information processing systems, 2015, 28: 1117-1125.

[18] Yin B, Corradi F, Bohté S M. Effective and efficient computation with multiple-timescale spiking recurrent neural networks[C]//International Conference on Neuromorphic Systems 2020. 2020: 1-8.



The primitive function is defined by

The function is used in[???].

static spiking_function(*x*, *alpha*)

static primitive_function(*x: torch.Tensor*, *alpha*)

training: bool

class spikingjelly.clock_driven.surrogate.**piecewise_leaky_relu**

基类：torch.autograd.function.Function

static forward(*ctx*, *x: torch.Tensor*, *w=1*, *c=0.01*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**PiecewiseLeakyReLU**(*w=1*, *c=0.01*, *spiking=True*)

基类：torch.nn.modules.module.Module

- *API in English*

参数

- **w** –w <= x <= w 时反向传播的梯度为 1 / 2w
- **c** –x > w 或 x < -w 时反向传播的梯度为 c

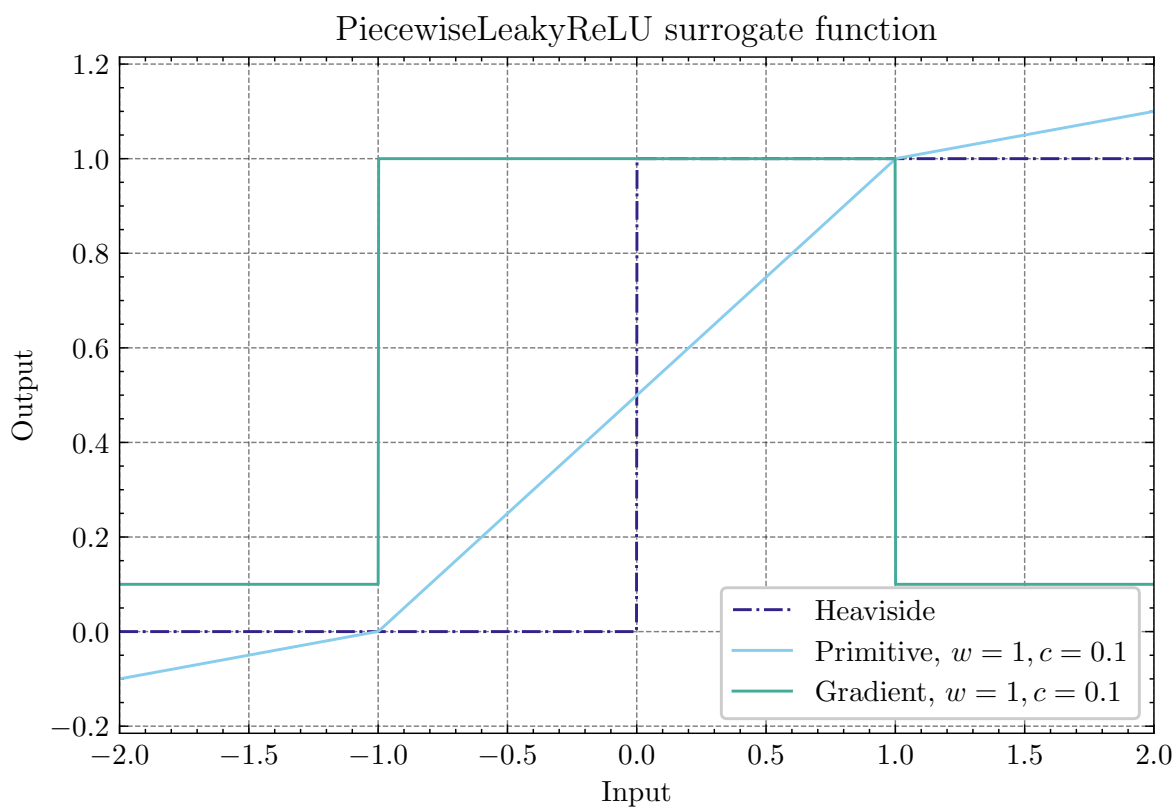
- **spiking** –是否输出脉冲，默认为 `True`，在前向传播时使用 `heaviside` 而在反向传播使用替代梯度。若为 `False` 则不使用替代梯度，前向传播时，使用反向传播时的梯度替代函数对应的原函数

分段线性的近似脉冲发放函数。梯度为

$$g'(x) = \begin{cases} \frac{1}{w}, & -w \leq x \leq w \\ c, & x < -w \ or \ x > w \end{cases}$$

对应的原函数为

$$g(x) = \begin{cases} cx + cw, & x < -w \\ \frac{1}{2w}x + \frac{1}{2}, & -w \leq x \leq w \\ cx - cw + 1, & x > w \end{cases}$$



该函数在文章[3?5910?1617] 中使用。

- *中文 API*

参数

- **w** –when `-w <= x <= w` the gradient is `1 / 2w`
- **c** –when `x > w` or `x < -w` the gradient is `c`
- **spiking** –whether output spikes. The default is `True` which means that using `heaviside` in forward propagation and using surrogate gradient in backward propagation. If `False`, in forward propagation, using the primitive function of the surrogate gradient function used in backward propagation

The piecewise surrogate spiking function. The gradient is defined by

$$g'(x) = \begin{cases} \frac{1}{w}, & -w \leq x \leq w \\ c, & x < -w \ or \ x > w \end{cases}$$

The primitive function is defined by

$$g(x) = \begin{cases} cx + cw, & x < -w \\ \frac{1}{2w}x + \frac{1}{2}, & -w \leq x \leq w \\ cx - cw + 1, & x > w \end{cases}$$

[3] Yin S, Venkataramanaiah S K, Chen G K, et al. Algorithm and hardware design of discrete-time spiking neural networks based on back propagation with binary activations[C]//2017 IEEE Biomedical Circuits and Systems Conference (BioCAS). IEEE, 2017: 1-5.

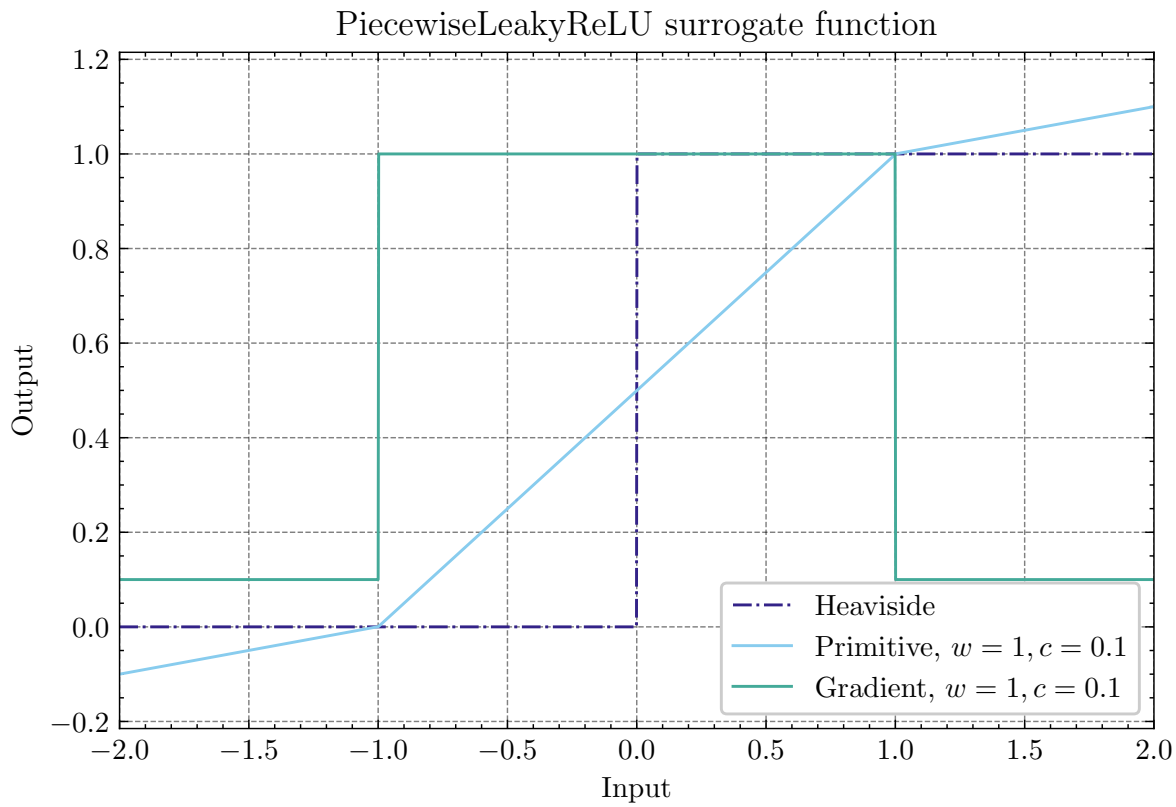
[5] Huh D, Sejnowski T J. Gradient descent for spiking neural networks[C]//Proceedings of the 32nd International Conference on Neural Information Processing Systems. 2018: 1440-1450.

[9] Wu Y, Deng L, Li G, et al. Direct training for spiking neural networks: Faster, larger, better[C]//Proceedings of the AAAI Conference on Artificial Intelligence. 2019, 33(01): 1311-1318.

[10] Gu P, Xiao R, Pan G, et al. STCA: Spatio-Temporal Credit Assignment with Delayed Feedback in Deep Spiking Neural Networks[C]//IJCAI. 2019: 1366-1372.

[16] Cheng X, Hao Y, Xu J, et al. LISNN: Improving Spiking Neural Networks with Lateral Interactions for Robust Object Recognition[C]//IJCAI. 1519-1525.

[17] Kaiser J, Mostafa H, Neftci E. Synaptic plasticity dynamics for deep continuous local learning (DECOLLE)[J]. Frontiers in Neuroscience, 2020, 14: 424.



The function is used in[????????].

forward(*x*)

static spiking_function(*x: torch.Tensor*, *w*, *c*)

static primitive_function(*x: torch.Tensor*, *w*, *c*)

training: bool

References

spikingjelly.clock_driven.ann2snn package

Subpackages

spikingjelly.clock_driven.ann2snn.examples package

Submodules

spikingjelly.clock_driven.ann2snn.examples.if_cnn_mnist module

Module contents

Submodules

spikingjelly.clock_driven.ann2snn.__init__ module

spikingjelly.clock_driven.ann2snn.modules module

Module contents

Module contents

6.5.2 spikingjelly.datasets package

Submodules

spikingjelly.datasets.asl_dvs module

class spikingjelly.datasets.asl_dvs.**ASLDVS**(*root: str, train: bool, split_ratio=0.9, use_frame=True, frames_num=10, split_by='number', normalization='max'*)

基类：*spikingjelly.datasets.utils.EventsFramesDatasetBase*

参数

- **root** (*str*) –保存数据集的根目录
- **train** (*bool*) –是否使用训练集
- **split_ratio** (*float*) –分割比例。每一类中前 split_ratio 的数据会被用作训练集，剩下的数据为测试集
- **use_frame** (*bool*) –是否将事件数据转换成帧数据
- **frames_num** (*int*) –转换后数据的帧数
- **split_by** (*str*) –脉冲数据转换成帧数据的累计方式。`'time'` 或 `'number'`
- **normalization** (*str or None*) –归一化方法，为 `None` 表示不进行归一化；为 `'frequency'` 则每一帧的数据除以每一帧的累加的原始数据数量；为 `'max'` 则每一帧的数据除以每一帧中数据的最大值；为 `norm` 则每一帧的数据减去每一帧中的均值，然后除以标准差

ASL-DVS 数据集，出自 Graph-Based Object Classification for Neuromorphic Vision Sensing，包含 24 个英文字母（从 A 到 Y，排除 J）的美国手语，American Sign Language (ASL)。更多信息参见 https://github.com/PIX2NVS/NVS2Graph，原始数据的下载地址为 https://www.dropbox.com/sh/ibq0jsicatn7l6r/AACNrNELV56rs1YInMWUs9CAa?dl=0。

关于转换成帧数据的细节，参见*integrate_events_to_frames()*。

static get_wh()

static download_and_extract(*download_root: str*, *extract_root: str*)

static read_bin(*file_name: str*)

static get_events_item(*file_name*)

static get_frames_item(*file_name*)

static create_frames_dataset(*events_data_dir: str*, *frames_data_dir: str*, *frames_num: int*, *split_by: str*, *normalization: str*)

spikingjelly.datasets.cifar10_dvs module

spikingjelly.datasets.cifar10_dvs.**read_bits**(*arr*, *mask=None*, *shift=None*)

spikingjelly.datasets.cifar10_dvs.**skip_header**(*fp*)

spikingjelly.datasets.cifar10_dvs.**load_raw_events**(*fp*, *bytes_skip=0*, *bytes_trim=0*, *filter_dvs=False*, *times_first=False*)

spikingjelly.datasets.cifar10_dvs.**parse_raw_address**(*addr*, *x_mask=4190208*, *x_shift=12*, *y_mask=2143289344*, *y_shift=22*, *polarity_mask=2048*, *polarity_shift=11*)

spikingjelly.datasets.cifar10_dvs.**load_events**(*fp*, *filter_dvs=False*, ***kwargs*)

class spikingjelly.datasets.cifar10_dvs.**CIFAR10DVS**(*root: str*, *train: bool*, *split_ratio=0.9*, *use_frame=True*, *frames_num=10*, *split_by='number'*, *normalization='max'*)

基类：`spikingjelly.datasets.utils.EventsFramesDatasetBase`

参数

- **root** (*str*) –保存数据集的根目录
- **train** (*bool*) –是否使用训练集
- **split_ratio** (*float*) –分割比例。每一类中前 split_ratio 的数据会被用作训练集，剩下的数据为测试集
- **use_frame** (*bool*) –是否将事件数据转换成帧数据
- **frames_num** (*int*) –转换后数据的帧数
- **split_by** (*str*) –脉冲数据转换成帧数据的累计方式。`'time'` 或 `'number'`
- **normalization** (*str or None*) –归一化方法，为 `None` 表示不进行归一化；为 `'frequency'` 则每一帧的数据除以每一帧的累加的原始数据数量；为 `'max'` 则每

一帧的数据除以每一帧中数据的最大值；为 norm 则每一帧的数据减去每一帧中的均值，然后除以标准差

CIFAR10 DVS 数据集，出自 CIFAR10-DVS: An Event-Stream Dataset for Object Classification，数据来源于 DVS 相机拍摄的显示器上的 CIFAR10 图片。原始数据的下载地址为 https://figshare.com/articles/dataset/CIFAR10-DVS_New/4724671。

关于转换成帧数据的细节，参见*integrate_events_to_frames()*。

static get_wh()

static download_and_extract(*download_root: str*, *extract_root: str*)

static read_bin(*file_name: str*)

static create_frames_dataset(*events_data_dir: str*, *frames_data_dir: str*, *frames_num: int*, *split_by: str*, *normalization: str*)

static get_frames_item(*file_name*)

static get_events_item(*file_name*)

spikingjelly.datasets.dvs_gesture module

class spikingjelly.datasets.dvs128_gesture.**DVS128Gesture**(*root: str*, *train: bool*, *use_frame=True*, *frames_num=10*, *split_by='number'*, *normalization='max'*)

基类：*spikingjelly.datasets.utils.EventsFramesDatasetBase*

参数

- **root** (*str*) –保存数据集的根目录。其中应该至少包含 *DvsGesture.tar.gz* 和 *gesture_mapping.csv*
- **train** (*bool*) –是否使用训练集
- **use_frame** (*bool*) –是否将事件数据转换成帧数据
- **frames_num** (*int*) –转换后数据的帧数
- **split_by** (*str*) –脉冲数据转换成帧数据的累计方式。'time' 或 'number'
- **normalization** (*str or None*) –归一化方法，为 None 表示不进行归一化；为 'frequency' 则每一帧的数据除以每一帧的累加的原始数据数量；为 'max' 则每一帧的数据除以每一帧中数据的最大值；为 norm 则每一帧的数据减去每一帧中的均值，然后除以标准差

DVS128 Gesture 数据集，出自 A Low Power, Fully Event-Based Gesture Recognition System，数据来源于 DVS 相机拍摄的手势。原始数据的原始下载地址参见 https://www.research.ibm.com/dvsgesture/。

关于转换成帧数据的细节，参见*integrate_events_to_frames()*。

参数

- **root** (*str*) –root directory of dataset, which should contain *DvsGesture.tar.gz* and *gesture_mapping.csv*
- **train** (*bool*) –whether use the train dataset. If *False*, use the test dataset
- **use_frame** (*bool*) –whether use the frames data. If *False*, use the events data
- **frames_num** (*int*) –the number of frames
- **split_by** (*str*) –how to split the events, can be `'number'`, `'time'`
- **normalization** (*str or None*) –how to normalize frames, can be `None`, `'frequency'`, `'max'`, `'norm'`, `'sum'`

DVS128 Gesture dataset, which is provided by *A Low Power, Fully Event-Based Gesture Recognition System <https://openaccess.thecvf.com/content_cvpr_2017/papers/Amir_A_Low_Power_CVPR_2017_paper.pdf>*, contains the gesture recorded by a DVS128 camera. The origin dataset can be downloaded from https://www.research.ibm.com/dvsgesture/.

For more details about converting events to frames, see *integrate_events_to_frames()*。

static get_wh()

static download_and_extract(*download_root: str*, *extract_root: str*)

static read_bin(*file_name: str*)

static convert_aedat_dir_to_npy_dir(*aedat_data_dir: str*, *events_npy_train_root: str*, *events_npy_test_root: str*)

static create_frames_dataset(*events_data_dir: str*, *frames_data_dir: str*, *frames_num: int*, *split_by: str*, *normalization: str*)

static get_events_item(*file_name*)

static get_frames_item(*file_name*)

spikingjelly.datasets.n_mnist module

class spikingjelly.datasets.n_mnist.**NMNIST**(*root: str*, *train: bool*, *use_frame=True*, *frames_num=10*, *split_by='number'*, *normalization='max'*)

基类：*spikingjelly.datasets.utils.EventsFramesDatasetBase*

参数

- **root** (*str*) –保存数据集的根目录
- **train** (*bool*) –是否使用训练集
- **use_frame** (*bool*) –是否将事件数据转换成帧数据
- **frames_num** (*int*) –转换后数据的帧数

- **split_by** (*str*) –脉冲数据转换成帧数据的累计方式。'time' 或 'number'
- **normalization** (*str or None*) –归一化方法，为 None 表示不进行归一化；为 'frequency' 则每一帧的数据除以每一帧的累加的原始数据数量；为 'max' 则每一帧的数据除以每一帧中数据的最大值；为 norm 则每一帧的数据减去每一帧中的均值，然后除以标准差

Neuromorphic-MNIST 数据集，出自 Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades，数据来源于 ATIS 相机拍摄的显示器上的 MNIST 图片。原始数据的原始下载地址参见 https://www.garrickorchard.com/datasets/n-mnist。

关于转换成帧数据的细节，参见*integrate_events_to_frames()*。

static get_wh()

static read_bin(*file_name: str*)

参数 **file_name** –N-MNIST 原始 bin 格式数据的文件名

返回 一个字典，键是 {‘t’，‘x’，‘y’，‘p’}，值是 np 数组

原始的 N-MNIST 提供的是 bin 格式数据，不能直接读取。本函数提供了一个读取的接口。本函数参考了 https://github.com/jackd/events-tfds 的代码。

原始数据以二进制存储：

Each example is a separate binary file consisting of a list of events. Each event occupies 40 bits as described below: bit 39 - 32: Xaddress (in pixels) bit 31 - 24: Yaddress (in pixels) bit 23: Polarity (0 for OFF, 1 for ON) bit 22 - 0: Timestamp (in microseconds)

static download_and_extract(*download_root: str*, *extract_root: str*)

static create_frames_dataset(*events_data_dir: str*, *frames_data_dir: str*, *frames_num: int*, *split_by: str*, *normalization: str*)

static get_events_item(*file_name*)

static get_frames_item(*file_name*)

spikingjelly.datasets.nav_gesture module

class spikingjelly.datasets.nav_gesture.**NAVGesture**(*root: str*, *use_frame=True*, *frames_num=10*, *split_by='number'*, *normalization='max'*)

基类：*spikingjelly.datasets.utils.EventsFramesDatasetBase*

参数

- **root** (*str*) –保存数据集的根目录
- **use_frame** (*bool*) –是否将事件数据转换成帧数据

- **frames_num** (*int*) –转换后数据的帧数
- **split_by** (*str*) –脉冲数据转换成帧数据的累计方式。`'time'` 或 `'number'`
- **normalization** (*str or None*) –归一化方法，为 `None` 表示不进行归一化；为 `'frequency'` 则每一帧的数据除以每一帧的累加的原始数据数量；为 `'max'` 则每一帧的数据除以每一帧中数据的最大值；为 `norm` 则每一帧的数据减去每一帧中的均值，然后除以标准差

NavGesture 数据集，出自 Event-based Visual Gesture Recognition with Background Suppression running on a smart-phone，数据来源于 ATIS 相机拍摄的手势。原始数据的原始下载地址参见 https://www.neuromorphic-vision.com/public/downloads/navgesture/。

关于转换成帧数据的细节，参见*integrate_events_to_frames()*。

static get_wh()

static read_bin(*file_name: str*)

参数 file_name –NavGesture 原始 bin 格式数据的文件名

返回 一个字典，键是 {‘t’，‘x’，‘y’，‘p’}，值是 np 数组

原始的 NavGesture 提供的是 bin 格式数据，不能直接读取。本函数提供了一个读取的接口。原始数据以二进制存储：

Events are encoded in binary format on 64 bits (8 bytes): 32 bits for timestamp 9 bits for x address 8 bits for y address 2 bits for polarity 13 bits padding

static get_label(*file_name*)

static get_events_item(*file_name*)

static get_frames_item(*file_name*)

static download_and_extract(*download_root: str*, *extract_root: str*)

static create_frames_dataset(*events_data_dir*, *frames_data_dir*, *frames_num=10*, *split_by='time'*, *normalization=None*)

spikingjelly.datasets.utils.module

class spikingjelly.datasets.utils.**FunctionThread**(*f*, **args*, ***kwargs*)

基类：`threading.Thread`

run()

spikingjelly.datasets.utils.**integrate_events_to_frames**(*events*, *height*, *width*, *frames_num=10*, *split_by='time'*, *normalization=None*)

- *API in English*

参数

- **events** –键是 {‘t’，‘x’，‘y’，‘p’}，值是 np 数组的的字典
- **height** –脉冲数据的高度，例如对于 CIFAR10-DVS 是 128
- **width** –脉冲数据的宽度，例如对于 CIFAR10-DVS 是 128
- **frames_num** –转换后数据的帧数
- **split_by** –脉冲数据转换成帧数据的累计方式，允许的取值为 `'number'`, `'time'`
- **normalization** –归一化方法，允许的取值为 `None`, `'frequency'`, `'max'`, `'norm'`, `'sum'`

返回 转化后的 frames 数据，是一个 `shape = [frames_num, 2, height, width]` 的 np 数组

记脉冲数据为 $E_i = (t_i, x_i, y_i, p_i), i = 0, 1, ..., N-1$，转换为帧数据 $F(j, p, x, y), j = 0, 1, ..., M-1$。

若划分方式 `split_by` 为 `'time'`，则

$$
\begin{aligned}
\Delta T &= [\frac{t_{N-1} - t_0}{M}] \\
j_l &= \arg\min_k \{t_k | t_k \geq t_0 + \Delta T \cdot j\} \\
j_r &= \begin{cases} \arg\max_k \{t_k | t_k < t_0 + \Delta T \cdot (j+1)\} + 1, & j < M-1 \\ N, & j = M-1 \end{cases} \\
F(j, p, x, y) &= \sum_{i=j_l}^{j_r - 1} \mathcal{I}_{\surd^{\S,\dagger}(\surd^{\rangle}, \S_{\rangle}, \dagger_{\rangle})}
\end{aligned}
$$

若划分方式 `split_by` 为 `'number'`，则

$$
\begin{aligned}
j_l &= [\frac{N}{M}] \cdot j \\
j_r &= \begin{cases} [\frac{N}{M}] \cdot (j+1), & j < M-1 \\ N, & j = M-1 \end{cases} \\
F(j, p, x, y) &= \sum_{i=j_l}^{j_r - 1} \mathcal{I}_{\surd^{\S,\dagger}(\surd^{\rangle}, \S_{\rangle}, \dagger_{\rangle})}
\end{aligned}
$$

其中 $\mathcal{I}$ 为示性函数，当且仅当 $(p, x, y) = (p_i, x_i, y_i)$ 时为 1，否则为 0。

若 `normalization` 为 `'frequency'`，

若 `split_by` 为 `time` 则

$$
F_{norm}(j, p, x, y) = \begin{cases} \frac{F(j,p,x,y)}{\Delta T}, & j < M-1 \\ \frac{F(j,p,x,y)}{\Delta T + (t_{N-1} - t_0) \bmod M}, & j = M-1 \end{cases}
$$

若 `split_by` 为 `number` 则

$$F_{norm}(j,p,x,y) = \frac{F(j,p,x,y)}{t_{j_r} - t_{j_l}}$$

若 `normalization` 为 `'max'` 则

$$F_{norm}(j,p,x,y) = \frac{F(j,p,x,y)}{\max F(j,p)}$$

若 `normalization` 为 `'norm'` 则

$$F_{norm}(j,p,x,y) = \frac{F(j,p,x,y) - \mathrm{E}(F(j,p))}{\sqrt{\mathrm{Var}(F(j,p))}}$$

若 `normalization` 为 `'sum'` 则

$$F_{norm}(j,p,x,y) = \frac{F(j,p,x,y)}{\sum_{a,b} F(j,p,a,b)}$$

- *中文 API*

参数

- **events** –a dict with keys are {'t', 'x', 'y', 'p'} and values are numpy arrays
- **height** –the height of events data, e.g., 128 for CIFAR10-DVS
- **width** –the width of events data, e.g., 128 for CIFAR10-DVS
- **frames_num** –frames number
- **split_by** –how to split the events, can be `'number'`, `'time'`
- **normalization** –how to normalize frames, can be `None`, `'frequency'`, `'max'`, `'norm'`, `'sum'`

返回 the frames data with `shape = [frames_num, 2, height, width]`

The events data are denoted by $E_i = (t_i, x_i, y_i, p_i), i = 0, 1, ..., N-1$, and the converted frames data are denoted by $F(j,p,x,y), j = 0, 1, ..., M-1$.

If `split_by` is `'time'`, then

$$\begin{aligned}
\Delta T &= [\frac{t_{N-1} - t_0}{M}] \\
j_l &= \arg\min_k \{t_k | t_k \geq t_0 + \Delta T \cdot j\} \\
j_r &= \begin{cases} \arg\max_k \{t_k | t_k < t_0 + \Delta T \cdot (j+1)\} + 1, & j < M-1 \\ N, & j = M-1 \end{cases} \\
F(j,p,x,y) &= \sum_{i=j_l}^{j_r - 1} \mathcal{I}_{\surd^{\S,\dagger}(\surd^{\rangle}, \S_{\rangle}, \dagger_{\rangle})}
\end{aligned}$$

If `split_by` is `'number'`, then

$$j_l = [\frac{N}{M}] \cdot j$$

$$j_r = \begin{cases} [\frac{N}{M}] \cdot (j+1), & j < M-1 \\ N, & j = M-1 \end{cases}$$

$$F(j,p,x,y) = \sum_{i=j_l}^{j_r-1} \mathcal{I}_{\surd^{\S,\dagger}(\surd^{\rangle},\S_{\rangle},\dagger_{\rangle})}$$

where $\mathcal{I}$ is the characteristic function， if and only if $(p,x,y) = (p_i, x_i, y_i)$, this function is identically 1 else 0.

If `normalization` is `'frequency'`,

if `split_by` is `time`,

$$F_{norm}(j,p,x,y) = \begin{cases} \frac{F(j,p,x,y)}{\Delta T}, & j < M-1 \\ \frac{F(j,p,x,y)}{\Delta T + (t_{N-1} - t_0) \bmod M}, & j = M-1 \end{cases}$$

if `split_by` is `number`,

$$F_{norm}(j,p,x,y) = \frac{F(j,p,x,y)}{t_{j_r} - t_{j_l}}$$

If `normalization` is `'max'`, then

$$F_{norm}(j,p,x,y) = \frac{F(j,p,x,y)}{\max F(j,p)}$$

If `normalization` is `'norm'`, then

$$F_{norm}(j,p,x,y) = \frac{F(j,p,x,y) - \mathrm{E}(F(j,p))}{\sqrt{\mathrm{Var}(F(j,p))}}$$

If `normalization` is `'sum'`, then

$$F_{norm}(j,p,x,y) = \frac{F(j,p,x,y)}{\sum_{a,b} F(j,p,a,b)}$$

`spikingjelly.datasets.utils.`**`normalize_frame`**(*frames: numpy.ndarray*, *normalization: str*)

`spikingjelly.datasets.utils.`**`convert_events_dir_to_frames_dir`**(*events_data_dir*, *frames_data_dir*, *suffix*, *read_function*, *height*, *width*, *frames_num=10*, *split_by='time'*, *normalization=None*, *thread_num=1*, *compress=False*)

spikingjelly.datasets.utils.**extract_zip_in_dir**(*source_dir*, *target_dir*)

参数

- **source_dir** –保存有 zip 文件的文件夹
- **target_dir** –保存 zip 解压后数据的文件夹

返回 None

将 source_dir 目录下的所有 *.zip 文件，解压到 target_dir 目录下的对应文件夹内

class spikingjelly.datasets.utils.**EventsFramesDatasetBase**

基类：torch.utils.data.dataset.Dataset

static get_wh()

返回

(width, height) width: int

events 或 frames 图像的宽度

height: int events 或 frames 图像的高度

返回类型 tuple

static read_bin(*file_name: str*)

参数 **file_name** (*str*) –脉冲数据的文件名

返回 events 键是 {‘t’，‘x’，‘y’，‘p’}，值是 np 数组的的字典

返回类型 dict

static get_events_item(*file_name*)

参数 **file_name** (*str*) –脉冲数据的文件名

返回

(events, label) events: dict

键是 {‘t’，‘x’，‘y’，‘p’}，值是 np 数组的的字典

label: int 数据的标签

返回类型 tuple

static get_frames_item(*file_name*)

参数 **file_name** (*str*) –帧数据的文件名

返回

(frames, label) frames: np.ndarray

shape = [frames_num, 2, height, width] 的 np 数组

label: int 数据的标签

返回类型 tuple

static download_and_extract(*download_root: str*, *extract_root: str*)

参数

- **download_root** (*str*) –保存下载文件的文件夹
- **extract_root** (*str*) –保存解压后文件的文件夹

下载数据集到 download_root，然后解压到 extract_root。

static create_frames_dataset(*events_data_dir: str*, *frames_data_dir: str*, *frames_num: int*, *split_by: str*, *normalization: str*)

参数

- **events_data_dir** (*str*) –保存脉冲数据的文件夹，文件夹的文件全部是脉冲数据
- **frames_data_dir** (*str*) –保存帧数据的文件夹
- **frames_num** (*int*) –转换后数据的帧数
- **split_by** (*str*) –脉冲数据转换成帧数据的累计方式
- **normalization** (*str or None*) –归一化方法

将 events_data_dir 文件夹下的脉冲数据全部转换成帧数据，并保存在 frames_data_dir。转换参数的详细含义，参见 integrate_events_to_frames 函数。

Module contents

class spikingjelly.datasets.**ASLDVS**(*root: str*, *train: bool*, *split_ratio=0.9*, *use_frame=True*, *frames_num=10*, *split_by='number'*, *normalization='max'*)

基类：*spikingjelly.datasets.utils.EventsFramesDatasetBase*

参数

- **root** (*str*) –保存数据集的根目录
- **train** (*bool*) –是否使用训练集
- **split_ratio** (*float*) –分割比例。每一类中前 split_ratio 的数据会被用作训练集，剩下的数据为测试集
- **use_frame** (*bool*) –是否将事件数据转换成帧数据
- **frames_num** (*int*) –转换后数据的帧数
- **split_by** (*str*) –脉冲数据转换成帧数据的累计方式。'time' 或 'number'

- **normalization** (*str or None*) –归一化方法，为 None 表示不进行归一化；为 'frequency' 则每一帧的数据除以每一帧的累加的原始数据数量；为 'max' 则每一帧的数据除以每一帧中数据的最大值；为 norm 则每一帧的数据减去每一帧中的均值，然后除以标准差

ASL-DVS 数据集，出自 Graph-Based Object Classification for Neuromorphic Vision Sensing，包含 24 个英文字母（从 A 到 Y，排除 J）的美国手语，American Sign Language (ASL)。更多信息参见 https://github.com/PIX2NVS/NVS2Graph，原始数据的下载地址为 https://www.dropbox.com/sh/ibq0jsicatn7l6r/AACNrNELV56rs1YInMWUs9CAa?dl=0。

关于转换成帧数据的细节，参见*integrate_events_to_frames()*。

static get_wh()

static download_and_extract(*download_root: str*, *extract_root: str*)

static read_bin(*file_name: str*)

static get_events_item(*file_name*)

static get_frames_item(*file_name*)

static create_frames_dataset(*events_data_dir: str*, *frames_data_dir: str*, *frames_num: int*, *split_by: str*, *normalization: str*)

class spikingjelly.datasets.**CIFAR10DVS**(*root: str*, *train: bool*, *split_ratio=0.9*, *use_frame=True*, *frames_num=10*, *split_by='number'*, *normalization='max'*)

基类：*spikingjelly.datasets.utils.EventsFramesDatasetBase*

参数

- **root** (*str*) –保存数据集的根目录
- **train** (*bool*) –是否使用训练集
- **split_ratio** (*float*) –分割比例。每一类中前 split_ratio 的数据会被用作训练集，剩下的数据为测试集
- **use_frame** (*bool*) –是否将事件数据转换成帧数据
- **frames_num** (*int*) –转换后数据的帧数
- **split_by** (*str*) –脉冲数据转换成帧数据的累计方式。'time' 或 'number'
- **normalization** (*str or None*) –归一化方法，为 None 表示不进行归一化；为 'frequency' 则每一帧的数据除以每一帧的累加的原始数据数量；为 'max' 则每一帧的数据除以每一帧中数据的最大值；为 norm 则每一帧的数据减去每一帧中的均值，然后除以标准差

CIFAR10 DVS 数据集，出自 CIFAR10-DVS: An Event-Stream Dataset for Object Classification，数据来源于 DVS 相机拍摄的显示器上的 CIFAR10 图片。原始数据的下载地址为 https://figshare.com/articles/dataset/CIFAR10-DVS_New/4724671。

关于转换成帧数据的细节，参见`integrate_events_to_frames()`。

static get_wh()

static download_and_extract(*download_root: str*, *extract_root: str*)

static read_bin(*file_name: str*)

static create_frames_dataset(*events_data_dir: str*, *frames_data_dir: str*, *frames_num: int*, *split_by: str*, *normalization: str*)

static get_frames_item(*file_name*)

static get_events_item(*file_name*)

class spikingjelly.datasets.**DVS128Gesture**(*root: str*, *train: bool*, *use_frame=True*, *frames_num=10*, *split_by='number'*, *normalization='max'*)

基类：`spikingjelly.datasets.utils.EventsFramesDatasetBase`

参数

- **root** (`str`) –保存数据集的根目录。其中应该至少包含 *DvsGesture.tar.gz* 和 *gesture_mapping.csv*
- **train** (`bool`) –是否使用训练集
- **use_frame** (`bool`) –是否将事件数据转换成帧数据
- **frames_num** (`int`) –转换后数据的帧数
- **split_by** (`str`) –脉冲数据转换成帧数据的累计方式。`'time'` 或 `'number'`
- **normalization** (`str or None`) –归一化方法，为 `None` 表示不进行归一化；为 `'frequency'` 则每一帧的数据除以每一帧的累加的原始数据数量；为 `'max'` 则每一帧的数据除以每一帧中数据的最大值；为 `norm` 则每一帧的数据减去每一帧中的均值，然后除以标准差

DVS128 Gesture 数据集，出自 A Low Power, Fully Event-Based Gesture Recognition System，数据来源于 DVS 相机拍摄的手势。原始数据的原始下载地址参见 https://www.research.ibm.com/dvsgesture/。

关于转换成帧数据的细节，参见`integrate_events_to_frames()`。

参数

- **root** (`str`) –root directory of dataset, which should contain *DvsGesture.tar.gz* and *gesture_mapping.csv*
- **train** (`bool`) –whether use the train dataset. If *False*, use the test dataset
- **use_frame** (`bool`) –whether use the frames data. If *False*, use the events data
- **frames_num** (`int`) –the number of frames
- **split_by** (`str`) –how to split the events, can be `'number', 'time'`

- **normalization** (*str or None*) –how to normalize frames, can be `None`, `'frequency'`, `'max'`, `'norm'`, `'sum'`

DVS128 Gesture dataset, which is provided by *A Low Power, Fully Event-Based Gesture Recognition System <https://openaccess.thecvf.com/content_cvpr_2017/papers/Amir_A_Low_Power_CVPR_2017_paper.pdf>*, contains the gesture recorded by a DVS128 camera. The origin dataset can be downloaded from https://www.research.ibm.com/dvsgesture/.

For more details about converting events to frames, see `integrate_events_to_frames()`。

static get_wh()

static download_and_extract(*download_root: str*, *extract_root: str*)

static read_bin(*file_name: str*)

static convert_aedat_dir_to_npy_dir(*aedat_data_dir: str*, *events_npy_train_root: str*, *events_npy_test_root: str*)

static create_frames_dataset(*events_data_dir: str*, *frames_data_dir: str*, *frames_num: int*, *split_by: str*, *normalization: str*)

static get_events_item(*file_name*)

static get_frames_item(*file_name*)

class spikingjelly.datasets.**NMNIST**(*root: str*, *train: bool*, *use_frame=True*, *frames_num=10*, *split_by='number'*, *normalization='max'*)

基类：`spikingjelly.datasets.utils.EventsFramesDatasetBase`

参数

- **root** (*str*) –保存数据集的根目录
- **train** (*bool*) –是否使用训练集
- **use_frame** (*bool*) –是否将事件数据转换成帧数据
- **frames_num** (*int*) –转换后数据的帧数
- **split_by** (*str*) –脉冲数据转换成帧数据的累计方式。`'time'` 或 `'number'`
- **normalization** (*str or None*) –归一化方法，为 `None` 表示不进行归一化；为 `'frequency'` 则每一帧的数据除以每一帧的累加的原始数据数量；为 `'max'` 则每一帧的数据除以每一帧中数据的最大值；为 `norm` 则每一帧的数据减去每一帧中的均值，然后除以标准差

Neuromorphic-MNIST 数据集，出自 Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades，数据来源于 ATIS 相机拍摄的显示器上的 MNIST 图片。原始数据的原始下载地址参见 https://www.garrickorchard.com/datasets/n-mnist。

关于转换成帧数据的细节，参见`integrate_events_to_frames()`。

static get_wh()

static read_bin(*file_name: str*)

参数 **file_name** –N-MNIST 原始 bin 格式数据的文件名

返回 一个字典，键是 {‘t’，‘x’，‘y’，‘p’}，值是 np 数组

原始的 N-MNIST 提供的是 bin 格式数据，不能直接读取。本函数提供了一个读取的接口。本函数参考了 https://github.com/jackd/events-tfds 的代码。

原始数据以二进制存储：

Each example is a separate binary file consisting of a list of events. Each event occupies 40 bits as described below: bit 39 - 32: Xaddress (in pixels) bit 31 - 24: Yaddress (in pixels) bit 23: Polarity (0 for OFF, 1 for ON) bit 22 - 0: Timestamp (in microseconds)

static download_and_extract(*download_root: str*, *extract_root: str*)

static create_frames_dataset(*events_data_dir: str*, *frames_data_dir: str*, *frames_num: int*, *split_by: str*, *normalization: str*)

static get_events_item(*file_name*)

static get_frames_item(*file_name*)

class spikingjelly.datasets.**NAVGesture**(*root: str*, *use_frame=True*, *frames_num=10*, *split_by='number'*, *normalization='max'*)

基类：*spikingjelly.datasets.utils.EventsFramesDatasetBase*

参数

- **root** (*str*) –保存数据集的根目录
- **use_frame** (*bool*) –是否将事件数据转换成帧数据
- **frames_num** (*int*) –转换后数据的帧数
- **split_by** (*str*) –脉冲数据转换成帧数据的累计方式。`'time'` 或 `'number'`
- **normalization** (*str or None*) –归一化方法，为 `None` 表示不进行归一化；为 `'frequency'` 则每一帧的数据除以每一帧的累加的原始数据数量；为 `'max'` 则每一帧的数据除以每一帧中数据的最大值；为 `norm` 则每一帧的数据减去每一帧中的均值，然后除以标准差

NavGesture 数据集，出自 Event-based Visual Gesture Recognition with Background Suppression running on a smart-phone，数据来源于 ATIS 相机拍摄的手势。原始数据的原始下载地址参见 https://www.neuromorphic-vision.com/public/downloads/navgesture/。

关于转换成帧数据的细节，参见*integrate_events_to_frames()*。

static get_wh()

static read_bin(*file_name: str*)

参数 **file_name** –NavGesture 原始 bin 格式数据的文件名

返回 一个字典，键是 {‘t’，‘x’，‘y’，‘p’}，值是 np 数组

原始的 NavGesture 提供的是 bin 格式数据，不能直接读取。本函数提供了一个读取的接口。原始数据以二进制存储：

Events are encoded in binary format on 64 bits (8 bytes): 32 bits for timestamp 9 bits for x address 8 bits for y address 2 bits for polarity 13 bits padding

static get_label(*file_name*)

static get_events_item(*file_name*)

static get_frames_item(*file_name*)

static download_and_extract(*download_root: str, extract_root: str*)

static create_frames_dataset(*events_data_dir, frames_data_dir, frames_num=10, split_by='time', normalization=None*)

class spikingjelly.datasets.**SPEECHCOMMANDS**(*label_dict: Dict, root: str, transform: Optional[Callable] = None, url: Optional[str] = 'speech_commands_v0.02', split: Optional[str] = 'train', folder_in_archive: Optional[str] = 'SpeechCommands', download: Optional[bool] = False*)

基类：`torch.utils.data.dataset.Dataset`

参数

- **label_dict** (*Dict*) –标签与类别的对应字典
- **root** (*str*) –数据集的根目录
- **transform** (*Callable, optional*) –A function/transform that takes in a raw audio
- **url** (*str, optional*) –数据集版本，默认为 v0.02
- **split** (*str, optional*) –数据集划分，可以是 `"train", "test", "val"`，默认为 `"train"`
- **folder_in_archive** (*str, optional*) –解压后的目录名称，默认为 `"SpeechCommands"`
- **download** (*bool, optional*) –是否下载数据，默认为 False

SpeechCommands 语音数据集，出自 Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition，根据给出的测试集与验证集列表进行了划分，包含 v0.01 与 v0.02 两个版本。

数据集包含三大类单词的音频：

1. 指令单词，共 10 个，" Yes"，"No"，"Up"，"Down"，"Left"，"Right"，"On"，"Off"，"Stop"，"Go"．对于 v0.02，还额外增加了 5 个：" Forward"，"Backward"，"Follow"，"Learn"，"Visual"．
2. 0~9 的数字，共 10 个：" One"，"Two"，"Three"，"Four"，"Five"，"Six"，"Seven"，"Eight"，"Nine"．

3. 非关键词，可以视为干扰词，共 10 个：”Bed”，“Bird”，“Cat”，“Dog”，“Happy”，“House”，“Marvin”，“Sheila”，“Tree”，“Wow”.

v0.01 版本包含共计 30 类，64,727 个音频片段，v0.02 版本包含共计 35 类，105,829 个音频片段。更详细的介绍参见前述论文，以及数据集的 README。

代码实现基于 torchaudio 并扩充了功能，同时也参考了 原论文的实现。

6.5.3 spikingjelly.event_driven package

spikingjelly.event_driven.examples package

Submodules

spikingjelly.event_driven.examples.tempotron_mnist module

class spikingjelly.event_driven.examples.tempotron_mnist.**Net**(*m*, *T*)

基类：torch.nn.modules.module.Module

forward(*x: torch.Tensor*)

training: bool

spikingjelly.event_driven.examples.tempotron_mnist.**main**()

返回 None

使用高斯调谐曲线编码器编码图像为脉冲，单层 Tempotron 进行 MNIST 识别。运行示例：

```
>>> import spikingjelly.event_driven.examples.tempotron_mnist as tempotron_mnist
>>> tempotron_mnist.main()
输入运行的设备，例如“cpu”或“cuda:0”
 input device, e.g., "cpu" or "cuda:0": cuda:15
输入保存 MNIST 数据集的位置，例如“./”
 input root directory for saving MNIST dataset, e.g., "./": ./mnist
输入 batch_size，例如“64”
 input batch_size, e.g., "64": 64
输入学习率，例如“1e-3”
 input learning rate, e.g., "1e-3": 1e-3
输入仿真时长，例如“100”
 input simulating steps, e.g., "100": 100
输入训练轮数，即遍历训练集的次数，例如“100”
 input training epochs, e.g., "100": 10
输入使用高斯调谐曲线编码每个像素点使用的神经元数量，例如“16”
 input neuron number for encoding a piexl in GaussianTuning encoder, e.g., "16":␣
↪16
输入保存 tensorboard 日志文件的位置，例如“./”
```

(下页继续)

(续上页)

```
 input root directory for saving tensorboard logs, e.g., "./": ./logs_tempotron_
↪mnist
cuda:15 ./mnist 64 0.001 100 100 16 ./logs_tempotron_mnist
train_acc 0.09375 0
cuda:15 ./mnist 64 0.001 100 100 16 ./logs_tempotron_mnist
train_acc 0.78125 512
...
```

Module contents

spikingjelly.event_driven.encoding package

Module contents

class spikingjelly.event_driven.encoding.**GaussianTuning**(*n*, *m*, *x_min: torch.Tensor*, *x_max: torch.Tensor*)

基类：object

参数

- **n** –特征的数量，int
- **m** –编码一个特征所使用的神经元数量，int
- **x_min** –n 个特征的最小值，shape=[n] 的 tensor
- **x_max** –n 个特征的最大值，shape=[n] 的 tensor

Bohte S M, Kok J N, La Poutre J A, et al. Error-backpropagation in temporally encoded networks of spiking neurons[J]. Neurocomputing, 2002, 48(1): 17-37. 中提出的高斯调谐曲线编码方式

编码器所使用的变量所在的 device 与 x_min.device 一致

encode(*x: torch.Tensor*, *max_spike_time=50*)

参数

- **x** –shape=[batch_size, n, k]，batch_size 个数据，每个数据含有 n 个特征，每个特征中有 k 个数据
- **max_spike_time** –最大（最晚）脉冲发放时间，也可以称为编码时间窗口的长度

返回 out_spikes, shape=[batch_size, n, k, m]，将每个数据编码成了 m 个神经元的脉冲发放时间

spikingjelly.event_driven.neuron package

Module contents

class spikingjelly.event_driven.neuron.**Tempotron**(*in_features*, *out_features*, *T*, *tau=15.0*, *tau_s=3.75*, *v_threshold=1.0*)

基类：torch.nn.modules.module.Module

参数

- **in_features** –输入数量，含义与 nn.Linear 的 in_features 参数相同
- **out_features** –输出数量，含义与 nn.Linear 的 out_features 参数相同
- **T** –仿真周期
- **tau** –LIF 神经元的积分时间常数
- **tau_s** –突触上的电流的衰减时间常数
- **v_threshold** –阈值电压

Gutig R, Sompolinsky H. The tempotron: a neuron that learns spike timing–based decisions[J]. Nature Neuroscience, 2006, 9(3): 420-428. 中提出的 Tempotron 模型

static psp_kernel(*t: torch.Tensor*, *tau*, *tau_s*)

参数

- **t** –表示时刻的 tensor
- **tau** –LIF 神经元的积分时间常数
- **tau_s** –突触上的电流的衰减时间常数

返回 t 时刻突触后的 LIF 神经元的电压值

static mse_loss(*v_max*, *v_threshold*, *label*, *num_classes*)

参数

- **v_max** –Tempotron 神经元在仿真周期内输出的最大电压值，与 forward 函数在 ret_type == ‘v_max’时的返回值相同。shape=[batch_size, out_features] 的 tensor
- **v_threshold** –Tempotron 的阈值电压，float 或 shape=[batch_size, out_features] 的 tensor
- **label** –样本的真实标签，shape=[batch_size] 的 tensor
- **num_classes** –样本的类别总数，int

返回 分类错误的神经元的电压，与阈值电压之差的均方误差

forward(*in_spikes: torch.Tensor*, *ret_type*)

参数 **in_spikes** –shape=[batch_size, in_features]

in_spikes[:, i] 表示第 i 个输入脉冲的脉冲发放时刻，介于 0 到 T 之间，T 是仿真时长

in_spikes[:, i] < 0 则表示无脉冲发放:param ret_type: 返回值的类项，可以为’v’,’v_max’,’spikes’ :return:

ret_type == ‘v’ : 返回一个 shape=[batch_size, out_features, T] 的 tensor，表示 out_features 个 Tempotron 神经元在仿真时长 T 内的电压值

ret_type == ‘v_max’ : 返回一个 shape=[batch_size, out_features] 的 tensor，表示 out_features 个 Tempotron 神经元在仿真时长 T 内的峰值电压

ret_type == ‘spikes’ : 返回一个 out_spikes，shape=[batch_size, out_features] 的 tensor，表示 out_features 个 Tempotron 神经元的脉冲发放时刻，out_spikes[:, i] 表示第 i 个输出脉冲的脉冲发放时刻，介于 0 到 T 之间，T 是仿真时长。out_spikes[:, i] < 0 表示无脉冲发放

training: bool

Module contents

6.5.4 spikingjelly.visualizing package

Module contents

spikingjelly.visualizing.**plot_2d_heatmap**(*array: numpy.ndarray, title: str, xlabel: str, ylabel: str, int_x_ticks=True, int_y_ticks=True, plot_colorbar=True, colorbar_y_label='magnitude', x_max=None, dpi=200*)

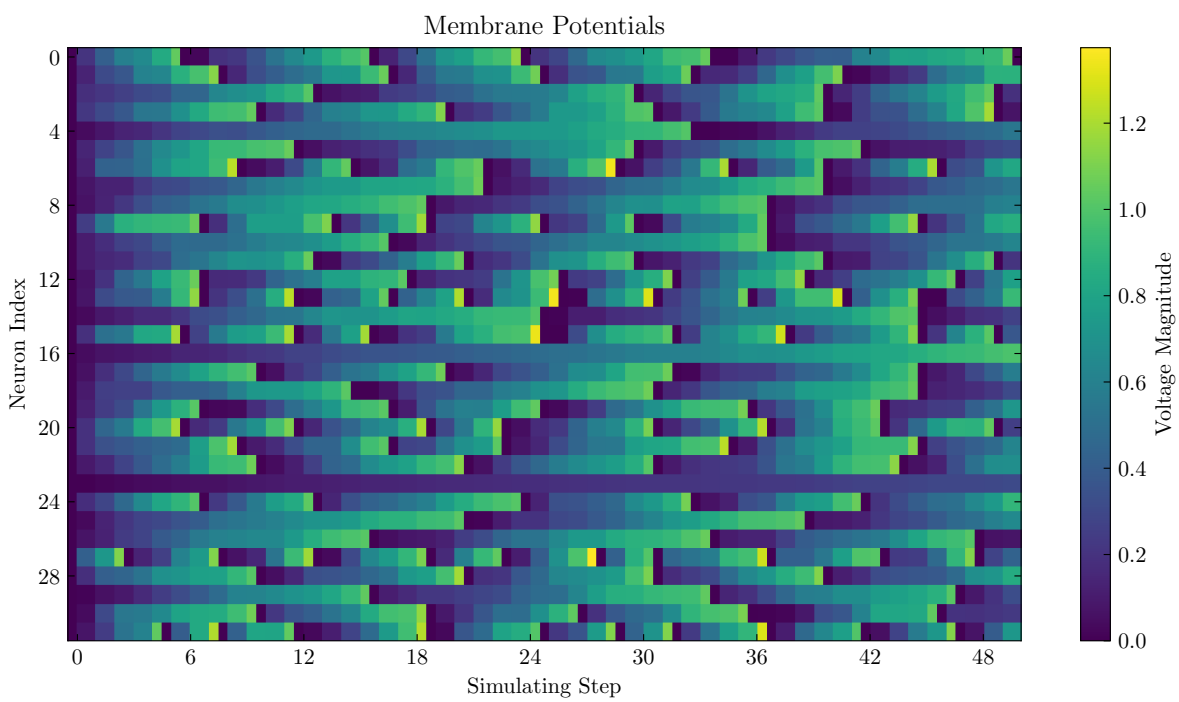
参数

- **array** –shape=[N, M] 的任意数组
- **title** –热力图的标题
- **xlabel** –热力图的 x 轴的 label
- **ylabel** –热力图的 y 轴的 label
- **int_x_ticks** –x 轴上是否只显示整数刻度
- **int_y_ticks** –y 轴上是否只显示整数刻度
- **plot_colorbar** –是否画出显示颜色和数值对应关系的 colorbar
- **colorbar_y_label** –colorbar 的 y 轴 label
- **x_max** –横轴的最大刻度。若设置为 `None`，则认为横轴的最大刻度是 `array.shape[1]`
- **dpi** –绘图的 dpi

返回 绘制好的 figure

绘制一张二维的热力图。可以用来绘制一张表示多个神经元在不同时刻的电压的热力图，示例代码：

```
from matplotlib import pyplot as plt
import torch
from spikingjelly.clock_driven import neuron
from spikingjelly import visualizing
import numpy as np

plt.style.use(['science'])
neuron_num = 32
T = 50
lif_node = neuron.LIFNode(monitor_state=True)
w = torch.rand([neuron_num]) * 50
for t in range(T):
    lif_node(w * torch.rand(size=[neuron_num]))
v_t_array = np.asarray(lif_node.monitor['v']).T  # v_t_array[i][j] 表示神经元 i 在 j 时刻的电压值
visualizing.plot_2d_heatmap(array=v_t_array, title='Membrane Potentials', xlabel='Simulating Step',
                            ylabel='Neuron Index', int_x_ticks=True, int_y_ticks=True,
                            plot_colorbar=True, colorbar_y_label='Voltage Magnitude', x_max=T, dpi=200)
plt.show()
```



spikingjelly.visualizing.**plot_2d_bar_in_3d**(*array: numpy.ndarray, title: str, xlabel: str, ylabel: str, zlabel: str, int_x_ticks=True, int_y_ticks=True, int_z_ticks=False, dpi=200*)

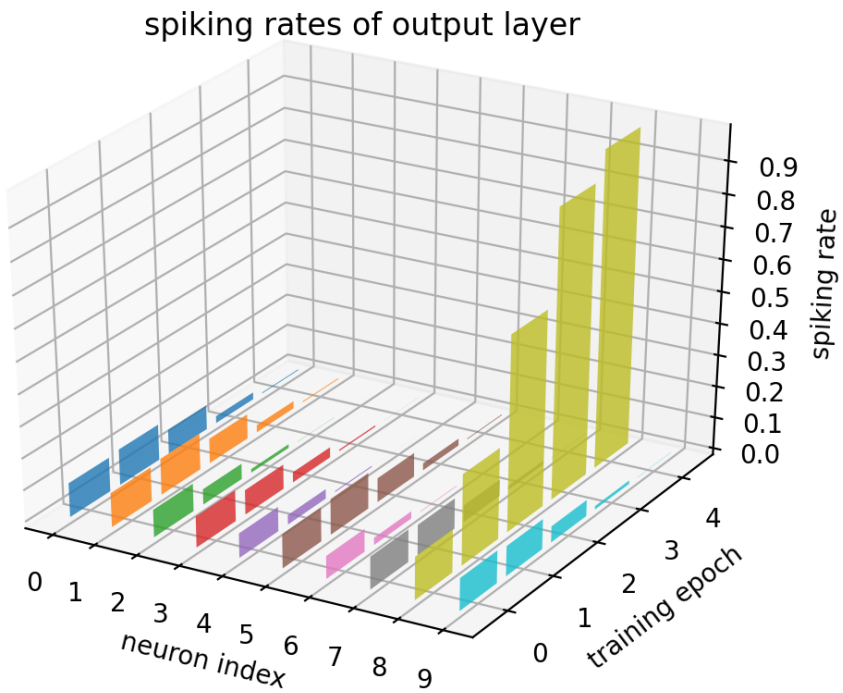
参数

- **array** –shape=[N, M] 的任意数组
- **title** –图的标题
- **xlabel** –x 轴的 label
- **ylabel** –y 轴的 label
- **zlabel** –z 轴的 label
- **int_x_ticks** –x 轴上是否只显示整数刻度
- **int_y_ticks** –y 轴上是否只显示整数刻度
- **int_z_ticks** –z 轴上是否只显示整数刻度
- **dpi** –绘图的 dpi

返回 绘制好的 figure

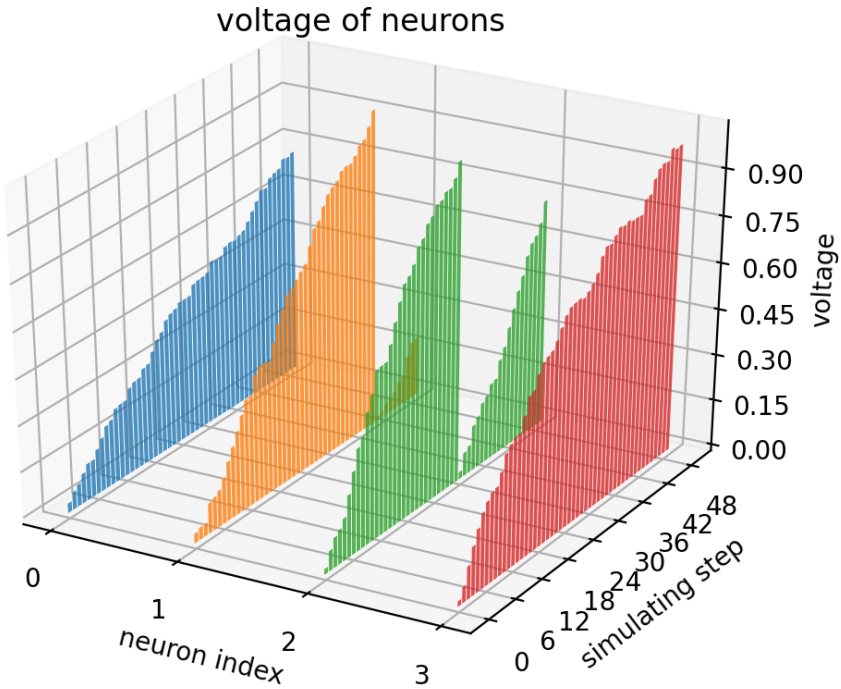
将 shape=[N, M] 的任意数组，绘制为三维的柱状图。可以用来绘制多个神经元的脉冲发放频率，随着时间的变化情况，示例代码：

```
Epochs = 5
N = 10
firing_rate = torch.zeros(N, Epochs)
init_firing_rate = torch.rand(size=[N])
for i in range(Epochs):
    firing_rate[:, i] = torch.softmax(init_firing_rate * (i + 1) ** 2, dim=0)
visualizing.plot_2d_bar_in_3d(firing_rate.numpy(), title='spiking rates of output␣
↪layer', xlabel='neuron index',
                              ylabel='training epoch', zlabel='spiking rate', int_
↪x_ticks=True, int_y_ticks=True,
                              int_z_ticks=False, dpi=200)
plt.show()
```



也可以用来绘制一张表示多个神经元在不同时刻的电压的热力图，示例代码：

```
neuron_num = 4
T = 50
lif_node = spikingjelly.event_driven.neuron.LIFNode(monitor=True)
w = torch.rand([neuron_num]) * 10
for t in range(T):
    lif_node(w * torch.rand(size=[neuron_num]))
v_t_array = np.asarray(lif_node.monitor['v']).T  # v_t_array[i][j] 表示神经元 i 在 j
时刻的电压值
visualizing.plot_2d_bar_in_3d(v_t_array, title='voltage of neurons', xlabel=
↪'neuron index',
                              ylabel='simulating step', zlabel='voltage', int_x_
↪ticks=True, int_y_ticks=True,
                              int_z_ticks=False, dpi=200)
plt.show()
```



spikingjelly.visualizing.**plot_1d_spikes**(*spikes: numpy.asarray, title: str, xlabel: str, ylabel: str, int_x_ticks=True, int_y_ticks=True, plot_firing_rate=True, firing_rate_map_title='Firing Rate', dpi=200*)

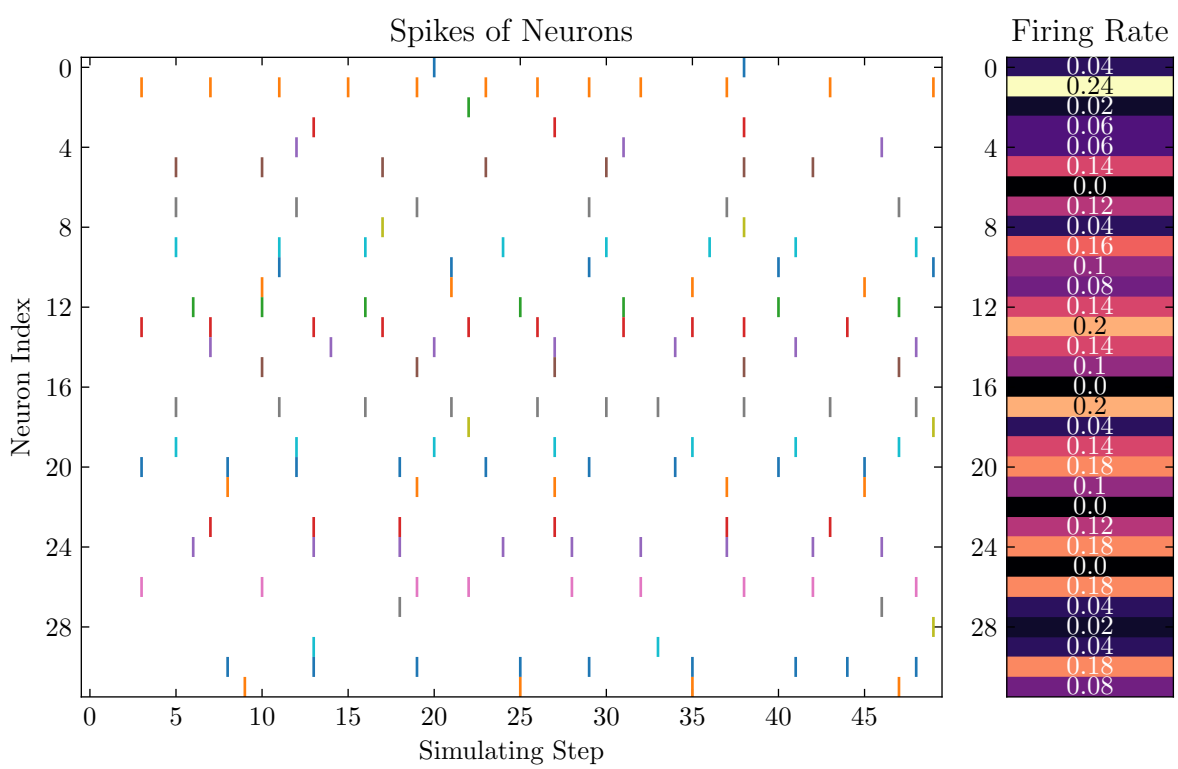
参数

- **spikes** –shape=[N, T] 的 np 数组，其中的元素只为 0 或 1，表示 N 个时长为 T 的脉冲数据
- **title** –热力图的标题
- **xlabel** –热力图的 x 轴的 label
- **ylabel** –热力图的 y 轴的 label
- **int_x_ticks** –x 轴上是否只显示整数刻度
- **int_y_ticks** –y 轴上是否只显示整数刻度
- **plot_firing_rate** –是否画出各个脉冲发放频率
- **firing_rate_map_title** –脉冲频率发放图的标题
- **dpi** –绘图的 dpi

返回 绘制好的 figure

画出 N 个时长为 T 的脉冲数据。可以用来画 N 个神经元在 T 个时刻的脉冲发放情况，示例代码：

```
from matplotlib import pyplot as plt
import torch
from spikingjelly.clock_driven import neuron
from spikingjelly import visualizing
import numpy as np

neuron_num = 32
T = 50
lif_node = neuron.LIFNode(monitor_state=True)
w = torch.rand([neuron_num]) * 50
for t in range(T):
    lif_node(w * torch.rand(size=[neuron_num]))
s_t_array = np.asarray(lif_node.monitor['s']).T  # s_t_array[i][j] 表示神经元 i 在 j 时刻释放的脉冲，为 0 或 1
visualizing.plot_1d_spikes(spikes=s_t_array, title='Spikes of Neurons', xlabel='Simulating Step',
                           ylabel='Neuron Index', int_x_ticks=True, int_y_ticks=True,
                           plot_firing_rate=True, firing_rate_map_title='Firing Rate', dpi=200)
plt.show()
```



spikingjelly.visualizing.**plot_2d_spiking_feature_map**(*spikes: numpy.asarray, nrows, ncols, space, title: str, dpi=200*)

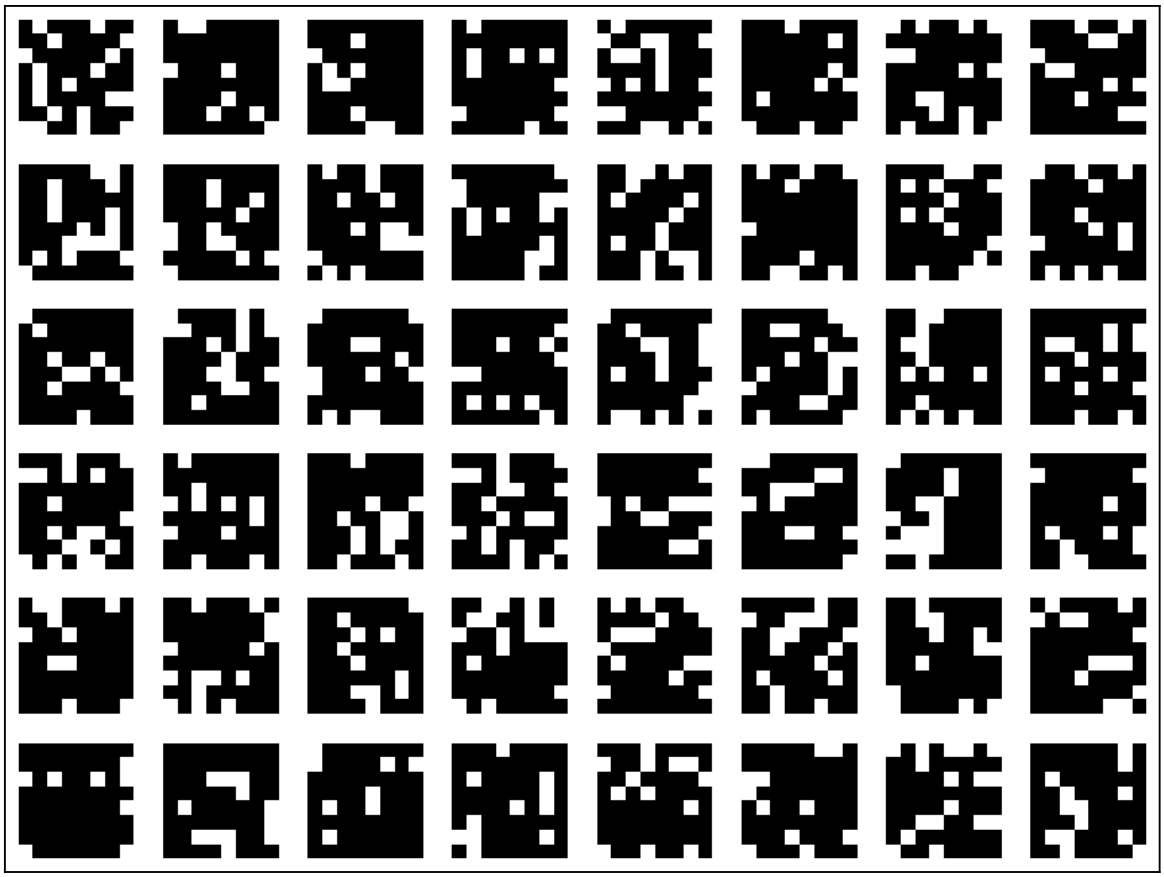
参数

- **spikes** –shape=[C, W, H]，C 个尺寸为 W * H 的脉冲矩阵，矩阵中的元素为 0 或 1。这样的矩阵一般来源于卷积层后的脉冲神经元的输出
- **nrows** –画成多少行
- **ncols** –画成多少列
- **space** –矩阵之间的间隙
- **title** –图的标题
- **dpi** –绘图的 dpi

返回 一个 figure，将 C 个矩阵全部画出，然后排列成 nrows 行 ncols 列

将 C 个尺寸为 W * H 的脉冲矩阵，全部画出，然后排列成 nrows 行 ncols 列。这样的矩阵一般来源于卷积层后的脉冲神经元的输出，通过这个函数可以对输出进行可视化。示例代码：

```
from spikingjelly import visualizing
import numpy as np

C = 48
W = 8
H = 8
spikes = (np.random.rand(C, W, H) > 0.8).astype(float)
visualizing.plot_2d_spiking_feature_map(spikes=spikes, nrows=6, ncols=8, space=2,␣
↪title='Spiking Feature Maps', dpi=200)
plt.show()
```



spikingjelly.visualizing.**plot_one_neuron_v_s**(*v: list, s: list, v_threshold=1.0, v_reset=0.0, title='V_{t} and S_{t} of the neuron', dpi=200*)

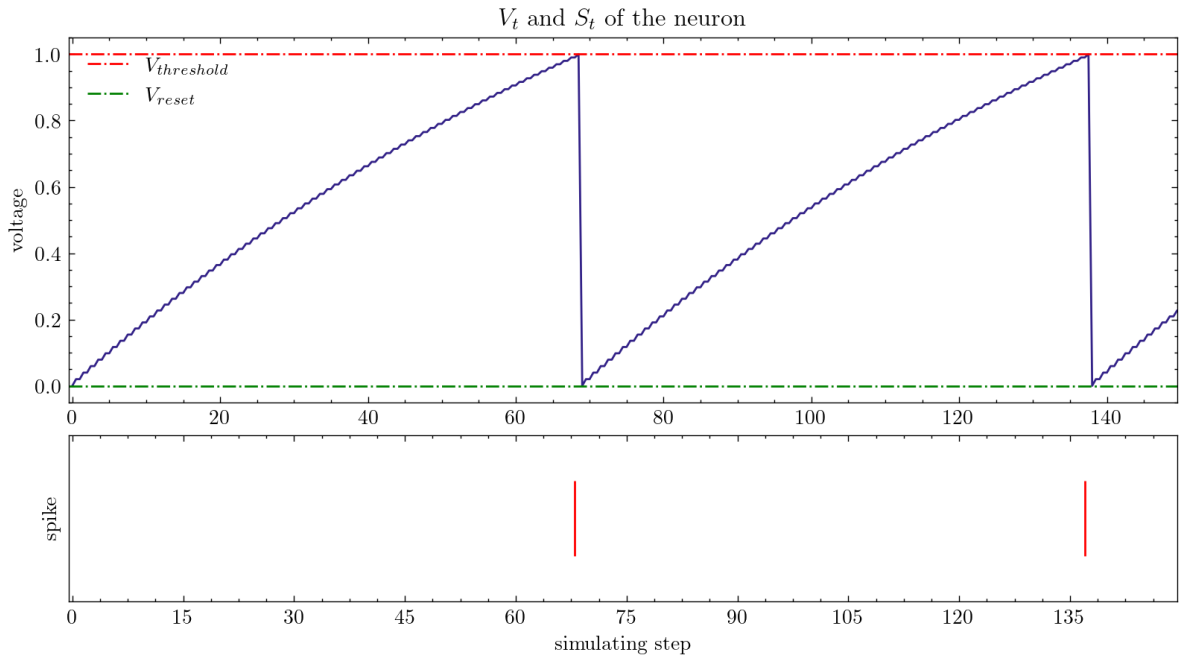
参数

- **v** –一个 `list`，存放神经元不同时刻的电压
- **s** –一个 `list`，存放神经元不同时刻释放的脉冲
- **v_threshold** –神经元的阈值电压
- **v_reset** –神经元的重置电压。也可以为 `None`
- **title** –图的标题
- **dpi** –绘图的 dpi

返回 一个 figure

绘制单个神经元的电压、脉冲随着时间的变化情况。常见的用法是，使用神经元的 `monitor` 记录的输入作为输入。示例代码：

```
lif = neuron.LIFNode()
lif.set_monitor(True)

x = torch.Tensor([2.0])
T = 150
for t in range(T):
    lif(x)
plot_one_neuron_v_s(lif.monitor['v'], lif.monitor['s'], v_threshold=lif.v_
↪threshold, v_reset=lif.v_reset, dpi=200)
plt.show()
```



6.5.5 spikingjelly.cext package

spikingjelly.cext.functional package

Module contents

class spikingjelly.cext.functional.**sparse_mm_dense_atf**

基类: torch.autograd.function.Function

static forward(*ctx*, *sparse: torch.Tensor*, *dense: torch.Tensor*)

static backward(*ctx*, *grad_output*)

spikingjelly.cext.functional.**sparse_mm_dense**(*sparse: torch.Tensor*, *dense: torch.Tensor*)

- *API in English*

参数

- **sparse** (*torch.Tensor*) –稀疏 2D tensor
- **dense** (*torch.Tensor*) –稠密 2D tensor

返回 sparse 和 dense 的矩阵乘

返回类型 torch.Tensor

对输入的稀疏的二维矩阵 `sparse` 和稠密的二维矩阵 `dense` 进行矩阵乘法。

> **警告:** 代码内部的实现方式是，首先将 `sparse` 转换为稀疏矩阵格式，然后再调用相关库进行运算。如果 `sparse` 不够稀疏，则该函数的速度会比普通矩阵乘法 `torch.mm` 慢很多。

> **警告:** 稀疏矩阵的乘法存在一定的计算误差，但误差并不显著，或可忽略。

> **警告:** 本函数不支持 CPU。

- *中文 API*

参数

- **sparse** (*torch.Tensor*) –a 2D sparse tensor
- **dense** (*torch.Tensor*) –a 2D dense tensor

返回 a matrix multiplication of the matrices `dense` and `sparse`

返回类型 torch.Tensor

Performs a matrix multiplication of the matrices `dense` and `sparse`.

Warning

This function is implemented by converting `sparse` to a sparse format and doing a sparse matrix multiplication. If the sparsity of `sparse` is not high enough, the speed of this function will be slower than `torch.mm`.

Warning

There are some numeral errors when doing the sparse matrix multiplication. But the errors are not significant.

Warning

This function does not support to run on cpu.

spikingjelly.cext.layer package

Module contents

class spikingjelly.cext.layer.**SparseLinear**(*in_features: int, out_features: int, bias: bool = True*)

基类：torch.nn.modules.linear.Linear

- *API in English*

参数

- **in_features** (*int*) –输入的特征数量
- **out_features** (*int*) –输出的特征数量
- **bias** (*bool*) –若为 False，则本层不含有可学习的偏置项。默认为 True

适用于稀疏输入的全连接层。与 torch.nn.Linear 的行为几乎相同。

> **警告：** 代码内部的实现方式是，首先将 sparse 转换为稀疏矩阵格式，然后再调用相关库进行运算。如果 sparse 不够稀疏，则该函数的速度会比普通矩阵乘法 torch.mm 慢很多。

> **警告：** 稀疏矩阵的乘法存在一定的计算误差，但误差并不显著，或可忽略。

> **警告：** 本层不支持 CPU。

- *中文 API*

参数

- **in_features** (*int*) –size of each input sample
- **out_features** (*int*) –size of each output sample
- **bias** (*bool*) –If set to False, the layer will not learn an additive bias. Default: True

The fully connected layer for sparse inputs. This module has a similar behavior as `torch.nn.Linear`.

Warning

This function is implemented by converting `sparse` to a sparse format and doing a sparse matrix multiplication. If the sparsity of `sparse` is not high enough, the speed of this function will be slower than `torch.mm`.

Warning

There are some numeral errors when doing the sparse matrix multiplication. But the errors are not significant.

Warning

This layer does not support to run on cpu.

forward(*sparse: torch.Tensor*) → torch.Tensor

in_features: int

out_features: int

weight: torch.Tensor

class spikingjelly.cext.layer.**AutoSparseLinear**(*in_features: int, out_features: int, bias: bool = True, in_spikes: bool = False*)

基类：`torch.nn.modules.linear.Linear`

- *API in English*

参数

- **in_features** (*int*) –输入的特征数量
- **out_features** (*int*) –输出的特征数量
- **bias** (*bool*) –若为 `False`，则本层不含有可学习的偏置项。默认为 `True`
- **in_spikes** (*bool*) –输入是否为脉冲，即元素均为 0 或 1

智能稀疏全连接层。对于任意输入，若它的 `batch_size` 对应的临界稀疏度未知，本层会首先运行基准测试*AutoSparseLinear.benchmark* 来获取临界稀疏度。临界稀疏度定义为，当输入是这一稀疏度时，稀疏矩阵乘法和普通矩阵乘法的速度恰好相同。对于任意输入，若它的 `batch_size` 对应的临界稀疏度已知，本层都会根据当前输入的稀疏度来智能决定是使用稀疏矩阵乘法还是普通矩阵乘法。

警告： 稀疏矩阵的乘法存在一定的计算误差，但误差并不显著，或可忽略。

警告： 稀疏矩阵乘法不支持 CPU。在 CPU 上运行，只会使用普通矩阵乘法。

- *中文 API*

参数

- **in_features** (*int*) –size of each input sample
- **out_features** (*int*) –size of each output sample
- **bias** (*bool*) –If set to `False`, the layer will not learn an additive bias. Default: `True`
- **in_spikes** (*bool*) –Whether inputs are spikes, whose elements are 0 and 1 Default: `False`

The auto sparse fully connected layer. For an input, if the corresponding critical sparsity of the input' s batch size is unknown, this layer will firstly run the benchmark *AutoSparseLinear.benchmark* to get the critical sparsity. The critical sparsity is the sparsity where the sparse matrix multiplication and the dense matrix multiplication have the same speed. For an input, if the corresponding critical sparsity of the input' s batch size is known, this layer can auto select whether using the sparse or dense matrix multiplication according to the current input' s sparsity.

Warning

There are some numeral errors when doing the sparse matrix multiplication. But the errors are not significant.

Warning

This sparse matrix multiplication does not support to run on cpu. When this layer is on CPU, the dense matrix multiplication will be always used.

---

forward(*x: torch.Tensor*) → torch.Tensor

extra_repr() → str

benchmark(*batch_size: int*, *device=None*, *run_times=1024*, *precision=0.0001*, *verbose=True*)

- *API in English*

参数

- **batch_size** (*int*) –输入的 batch size

- **device** (*str or None*) –运行基准测试所在的设备。若为 `None`，则会被设置成本层所在的设备。
- **run_times** (*int*) –运行稀疏/普通矩阵乘法的重复实验的次数。越大，则基准测试的结果越可靠
- **precision** (*float*) –二分搜索的最终临界稀疏值的精度
- **verbose** (*bool*) –是否打印出测试过程中的日志

使用二分查找，在输入的 batch size 为 `batch_size` 时，在每个稀疏度上重复运行 `run_times` 次稀疏/普通矩阵乘法，比较两者的速度，直到搜索到临界稀疏度。若搜索达到精度范围 `precision` 时，普通矩阵乘法仍然比稀疏矩阵乘法快，则会将临界稀疏度设置成 `None`。

- *中文 API*

参数

- **batch_size** (*int*) –batch size of the input
- **device** (*str*) –where to running the benchmark. If `None`, it will be set as same with this layer’s device
- **run_times** (*int*) –the number of replicated running times for sparse/dense matrix multiplication. The benchmark result will be more reliable with a larger `run_times`
- **precision** (*float*) –the precision of binary searching critical sparsity
- **verbose** (*bool*) –If `True`, this function will print logs during running

Using the binary search to find the critical sparsity when the batch size of the input is `batch_size`. This function will run `run_times` sparse/dense matrix multiplication on different sparsity and compare their speeds until it finds the cirtical sparsity. If the dense matrix multiplication is faster than the sparse matrix multiplication when searching exceeds `precision`, then the critical sparsity will be set to `None`.

in_features: *int*

out_features: *int*

weight: *torch.Tensor*

spikingjelly.cext.neuron package

Module contents

spikingjelly.cext.neuron.**hard_reset_forward_template**(*x: torch.Tensor, v: torch.Tensor, v_threshold: float, v_reset: float, *args, **kwargs*)

- *API in English*

参数

- **x** (*torch.Tensor*) – X_t
- **v** (*torch.Tensor*) – V_{t-1}
- **v_threshold** (*float*) – $V_{threshold}$
- **v_reset** (*float*) – V_{reset}

返回 (spike, v_next)，其中 spike 是 S_t，*v_next* 是 V_t

返回类型 tuple

对神经元进行单步的电压更新，其中电压重置方式是硬重置 (hard reset)。更新的方程为

$$
\begin{aligned}
H_t &= f(X_t, V_{t-1}; \theta) \\
S_t &= \Theta(H_t - V_{threshold}) \\
V_t &= S_t V_{reset} + (1 - S_t) H_t
\end{aligned}
$$

其中 $f(\cdot)$ 是充电方程，θ 是神经元自身的参数。

- *中文 API*

参数

- **x** (*torch.Tensor*) – X_t
- **v** (*torch.Tensor*) – V_{t-1}
- **v_threshold** (*float*) – $V_{threshold}$
- **v_reset** (*float*) – V_{reset}

返回 (spike, v_next), where spike is S_t, and v_next is V_t

返回类型 tuple

Update the membrane potential of the neuron by one time step with hard reset. The update is calculated by

$$
\begin{aligned}
H_t &= f(X_t, V_{t-1}; \theta) \\
S_t &= \Theta(H_t - V_{threshold}) \\
V_t &= S_t V_{reset} + (1 - S_t) H_t
\end{aligned}
$$

where $f(\cdot)$ is the charging equation and θ is the neuron' s parameters.

spikingjelly.cext.neuron.**hard_reset_fptt_template**(*x_seq: torch.Tensor, v: torch.Tensor, v_threshold: float, v_reset: float, *args, **kwargs*)

- *API in English*

参数

- **x** (*torch.Tensor*) – $X_t, t = 0, 1, ..., T-1$
- **v** (*torch.Tensor*) – V_{-1}
- **v_threshold** (*float*) – $V_{threshold}$
- **v_reset** (*float*) – V_{reset}

返回 (spike_seq, v_next)，其中 spike 是 $S_t, t = 0, 1, ..., T-1$，*v_next* 是 V_{T-1}

返回类型 tuple

hard_reset_forward_template 的多步版本。

- *中文 API*

参数

- **x** (*torch.Tensor*) – $X_t, t = 0, 1, ..., T-1$
- **v** (*torch.Tensor*) – V_{-1}
- **v_threshold** (*float*) – $V_{threshold}$
- **v_reset** (*float*) – V_{reset}

返回 (spike_seq, v_next), where spike is $S_t, t = 0, 1, ..., T-1$, *v_next* is V_{T-1}

返回类型 tuple

The multi-step version of *hard_reset_forward_template*.

spikingjelly.cext.neuron.**soft_reset_forward_template**(*x: torch.Tensor*, *v: torch.Tensor*, *v_threshold: float*, **args*, ***kwargs*)

- *API in English*

参数

- **x** (*torch.Tensor*) – X_t
- **v** (*torch.Tensor*) – V_{t-1}
- **v_threshold** (*float*) – $V_{threshold}$

返回 (spike, v_next)，其中 spike 是 S_t，*v_next* 是 V_t

返回类型 tuple

对神经元进行单步的电压更新，其中电压重置方式是软重置 (soft reset)。更新的方程为

$$
\begin{aligned}
H_t &= f(X_t, V_{t-1}; \theta) \\
S_t &= \Theta(H_t - V_{threshold}) \\
V_t &= H_t - S_t V_{threshold}
\end{aligned}
$$

其中 $f(\cdot)$ 是充电方程，θ 是神经元自身的参数。

- 中文 API

参数

- **x** (torch.Tensor) – X_t
- **v** (torch.Tensor) – V_{t-1}
- **v_threshold** (float) – $V_{threshold}$

返回 (spike, v_next), where spike is S_t, and v_next is V_t

返回类型 tuple

Update the membrane potential of the neuron by one time step with soft reset. The update is calculated by

$$H_t = f(X_t, V_{t-1}; \theta)$$
$$S_t = \Theta(H_t - V_{threshold})$$
$$V_t = H_t - S_t V_{threshold}$$

where $f(\cdot)$ is the charging equation and θ is the neuron' s parameters.

spikingjelly.cext.neuron.**soft_reset_fptt_template**(*x_seq: torch.Tensor, v: torch.Tensor, v_threshold: float, *args, **kwargs*)

- API in English

参数

- **x** (torch.Tensor) – $X_t, t = 0, 1, ..., T-1$
- **v** (torch.Tensor) – V_{-1}
- **v_threshold** (float) – $V_{threshold}$

返回 (spike_seq, v_next)，其中 spike 是 $S_t, t = 0, 1, ..., T-1$，*v_next* 是 V_{T-1}

返回类型 tuple

soft_reset_forward_template 的多步版本。

- 中文 API

参数

- **x** (torch.Tensor) – $X_t, t = 0, 1, ..., T-1$
- **v** (torch.Tensor) – V_{-1}
- **v_threshold** (float) – $V_{threshold}$

返回 (spike_seq, v_next), where spike is $S_t, t = 0, 1, ..., T-1$, *v_next* is V_{T-1}

返回类型 tuple

The multi-step version of *soft_reset_forward_template*.

spikingjelly.cext.neuron.**hard_reset_forward_with_grad_template**(*x: torch.Tensor*, *v: torch.Tensor*, *v_threshold: float*, *v_reset: float*, *alpha: float*, *detach_reset: bool*, *grad_surrogate_function_index: int*, **args*, ***kwargs*)

- *API in English*

参数

- **x** (*torch.Tensor*) –X_t
- **v** (*torch.Tensor*) –V_{t-1}
- **v_threshold** (*float*) –$V_{threshold}$
- **v_reset** (*float*) –V_{reset}
- **alpha** (*float*) –α
- **detach_reset** (*bool*) –是否在反向传播的计算图中断开重置过程
- **grad_surrogate_function_index** (*int*) –梯度替代函数的索引

返回 (spike, v_next, grad_s_to_h, grad_v_to_h)，其中 spike 是 S_t，*v_next* 是 V_t，grad_s_to_h 是 $\frac{\partial S_t}{\partial H_t}$，grad_v_to_h 是 $\frac{\partial V_t}{\partial H_t}$

返回类型 tuple

对神经元进行单步的电压更新，其中电压重置方式是硬重置 (hard reset)。更新的方程为

$$
\begin{aligned}
H_t &= f(X_t, V_{t-1}; \theta) \\
S_t &= \Theta(H_t - V_{threshold}) \\
V_t &= S_t V_{reset} + (1 - S_t) H_t
\end{aligned}
$$

其中 $f(\cdot)$ 是充电方程，θ 是神经元自身的参数。并且会计算出反向传播所需的梯度

$$
\begin{aligned}
\frac{\partial S_t}{\partial H_t} &= \Theta'(H_t - V_{threshold}) = \sigma(\alpha(H_t - V_{threshold})) \\
\frac{\partial V_t}{\partial H_t} &= 1 - S_t + (V_{reset} - H_t)\frac{\partial S_t}{\partial H_t}
\end{aligned}
$$

- *中文 API*

参数

- **x** (*torch.Tensor*) –X_t
- **v** (*torch.Tensor*) –V_{t-1}
- **v_threshold** (*float*) –$V_{threshold}$
- **v_reset** (*float*) –V_{reset}
- **alpha** (*float*) –α
- **detach_reset** (*bool*) –whether detach the neuronal reset during backward
- **grad_surrogate_function_index** (*int*) –index of the gradient surrogate function

返回 (spike, v_next, grad_s_to_h, grad_v_to_h), where spike is S_t, *v_next* is V_t, grad_s_to_h is $\frac{\partial S_t}{\partial H_t}$, grad_v_to_h is $\frac{\partial V_t}{\partial H_t}$

返回类型 tuple

Update the membrane potential of the neuron by one time step with hard reset. The update is calculated by

$$
\begin{aligned}
H_t &= f(X_t, V_{t-1}; \theta) \\
S_t &= \Theta(H_t - V_{threshold}) \\
V_t &= S_t V_{reset} + (1 - S_t) H_t
\end{aligned}
$$

where $f(\cdot)$ is the charging equation and θ is the neuron' s parameters. This function will also calculate the gradients which the backward function needs

$$
\begin{aligned}
\frac{\partial S_t}{\partial H_t} &= \Theta'(H_t - V_{threshold}) = \sigma(\alpha(H_t - V_{threshold})) \\
\frac{\partial V_t}{\partial H_t} &= 1 - S_t + (V_{reset} - H_t)\frac{\partial S_t}{\partial H_t}
\end{aligned}
$$

spikingjelly.cext.neuron.**hard_reset_fptt_with_grad_template**(*x_seq: torch.Tensor, v: torch.Tensor, v_threshold: float, v_reset: float, alpha: float, detach_reset: bool, grad_surrogate_function_index: int, *args, **kwargs*)

- *API in English*

参数

- **x_seq** –$X_t, t = 0, 1, ..., T-1$
- **v** (*torch.Tensor*) –V_{-1}
- **v_threshold** (*float*) –$V_{threshold}$
- **v_reset** (*float*) –V_{reset}

- **alpha** (*float*) –α
- **detach_reset** (*bool*) –是否在反向传播的计算图中断开重置过程
- **grad_surrogate_function_index** (*int*) –梯度替代函数的索引

返回 (spike_seq, v_next, grad_s_to_h, grad_v_to_h)，其中 spike_seq 是 $S_t, t = 0, 1, ..., T-1$，*v_next* 是 V_{T-1}，grad_s_to_h 是 $\frac{\partial S_t}{\partial H_t}, t = 0, 1, ..., T-1$，grad_v_to_h 是 $\frac{\partial V_t}{\partial H_t}, t = 0, 1, ..., T-1$

返回类型 tuple

hard_reset_forward_with_grad_template 的多步版本。

- *中文 API*

参数

- **x_seq** –$X_t, t = 0, 1, ..., T-1$
- **v** (*torch.Tensor*) –V_{-1}
- **v_threshold** (*float*) –$V_{threshold}$
- **v_reset** (*float*) –V_{reset}
- **alpha** (*float*) –α
- **detach_reset** (*bool*) –whether detach the neuronal reset during backward
- **grad_surrogate_function_index** (*int*) –index of the gradient surrogate function

返回 (spike_seq, v_next, grad_s_to_h, grad_v_to_h), where spike_seq is $S_t, t = 0, 1, ..., T-1$, *v_next* is V_{T-1}, grad_s_to_h is $\frac{\partial S_t}{\partial H_t}, t = 0, 1, ..., T-1$, grad_v_to_h is $\frac{\partial V_t}{\partial H_t}, t = 0, 1, ..., T-1$

返回类型 tuple

The multi-step version of *hard_reset_forward_with_grad_template*.

spikingjelly.cext.neuron.**soft_reset_forward_with_grad_template**(*x: torch.Tensor, v: torch.Tensor, v_threshold: float, alpha: float, detach_reset: bool, grad_surrogate_function_index: int, *args, **kwargs*)

- *API in English*

参数

- **x** (*torch.Tensor*) –X_t
- **v** (*torch.Tensor*) –V_{t-1}
- **v_threshold** (*float*) –$V_{threshold}$
- **alpha** (*float*) –α
- **detach_reset** (*bool*) –是否在反向传播的计算图中断开重置过程
- **grad_surrogate_function_index** (*int*) –梯度替代函数的索引

返回 (spike, v_next, grad_s_to_h, grad_v_to_h)，其中 spike 是 S_t，*v_next* 是 V_t，grad_s_to_h 是 $\frac{\partial S_t}{\partial H_t}$，grad_v_to_h 是 $\frac{\partial V_t}{\partial H_t}$

返回类型 tuple

对神经元进行单步的电压更新，其中电压重置方式是软重置 (soft reset)。更新的方程为

$$
\begin{aligned}
H_t &= f(X_t, V_{t-1}; \theta) \\
S_t &= \Theta(H_t - V_{threshold}) \\
V_t &= H_t - S_t V_{threshold}
\end{aligned}
$$

其中 $f(\cdot)$ 是充电方程，θ 是神经元自身的参数。并且会计算出反向传播所需的梯度

$$
\begin{aligned}
\frac{\partial S_t}{\partial H_t} &= \Theta'(H_t - V_{threshold}) = \sigma(\alpha(H_t - V_{threshold})) \\
\frac{\partial V_t}{\partial H_t} &= 1 - V_{threshold} \frac{\partial S_t}{\partial H_t}
\end{aligned}
$$

- *中文 API*

参数

- **x** (*torch.Tensor*) –X_t
- **v** (*torch.Tensor*) –V_{t-1}
- **v_threshold** (*float*) –$V_{threshold}$
- **alpha** (*float*) –α
- **detach_reset** (*bool*) –whether detach the neuronal reset during backward
- **grad_surrogate_function_index** (*int*) –index of the gradient surrogate function

返回 (spike, v_next, grad_s_to_h, grad_v_to_h), where spike is S_t, *v_next* is V_t, grad_s_to_h is $\frac{\partial S_t}{\partial H_t}$, grad_v_to_h is $\frac{\partial V_t}{\partial H_t}$

返回类型 tuple

Update the membrane potential of the neuron by one time step with soft reset. The update is calculated by

$$H_t = f(X_t, V_{t-1}; \theta)$$
$$S_t = \Theta(H_t - V_{threshold})$$
$$V_t = H_t - S_t V_{threshold}$$

where $f(\cdot)$ is the charging equation and θ is the neuron' s parameters. This function will also calculate the gradients which the backward function needs

$$\frac{\partial S_t}{\partial H_t} = \Theta'(H_t - V_{threshold}) = \sigma(\alpha(H_t - V_{threshold}))$$
$$\frac{\partial V_t}{\partial H_t} = 1 - V_{threshold}\frac{\partial S_t}{\partial H_t}$$

spikingjelly.cext.neuron.**soft_reset_fptt_with_grad_template**(*x_seq: torch.Tensor, v: torch.Tensor, v_threshold: float, alpha: float, detach_reset: bool, grad_surrogate_function_index: int, *args, **kwargs*)

- *API in English*

参数

- **x_seq** –$X_t, t = 0, 1, ..., T-1$
- **v** (*torch.Tensor*) –V_{-1}
- **v_threshold** (*float*) –$V_{threshold}$
- **alpha** (*float*) –α
- **detach_reset** (*bool*) –是否在反向传播的计算图中断开重置过程
- **grad_surrogate_function_index** (*int*) –梯度替代函数的索引

返回 (spike_seq, v_next, grad_s_to_h, grad_v_to_h)，其中 spike_seq 是 $S_t, t = 0, 1, ..., T-1$，*v_next* 是 V_{T-1}，grad_s_to_h 是 $\frac{\partial S_t}{\partial H_t}, t = 0, 1, ..., T-1$，grad_v_to_h 是 $\frac{\partial V_t}{\partial H_t}, t = 0, 1, ..., T-1$

返回类型 tuple

soft_reset_forward_with_grad_template 的多步版本。

- *中文 API*

参数

- **x_seq** –$X_t, t = 0, 1, ..., T-1$

- **v** (*torch.Tensor*) –V_{-1}
- **v_threshold** (*float*) –$V_{threshold}$
- **v_reset** (*float*) –V_{reset}
- **alpha** (*float*) –α
- **detach_reset** (*bool*) –whether detach the neuronal reset during backward
- **grad_surrogate_function_index** (*int*) –index of the gradient surrogate function

返回 (spike_seq, v_next, grad_s_to_h, grad_v_to_h), where spike_seq is $S_t, t = 0, 1, ..., T-1$, *v_next* is V_{T-1}, grad_s_to_h is $\frac{\partial S_t}{\partial H_t}, t = 0, 1, ..., T-1$, grad_v_to_h is $\frac{\partial V_t}{\partial H_t}, t = 0, 1, ..., T-1$

返回类型 tuple

The multi-step version of *soft_reset_forward_with_grad_template*.

spikingjelly.cext.neuron.**backward_template**(*grad_spike: torch.Tensor, grad_v_next: torch.Tensor, grad_s_to_h: torch.Tensor, grad_v_to_h: float, *args, **kwargs*)

- *API in English*

参数

- **grad_spike** (*torch.Tensor*) –$\frac{\partial L}{\partial S_t}$
- **grad_v_next** (*torch.Tensor*) –$\frac{\partial L}{\partial V_t}$
- **grad_s_to_h** (*torch.Tensor*) –$\frac{\partial S_t}{\partial H_t}$
- **grad_v_to_h** (*torch.Tensor*) –$\frac{\partial V_t}{\partial H_t}$

返回 (grad_x, grad_v)，其中 grad_x 是 $\frac{\partial L}{\partial X_t}$，grad_v 是 $\frac{\partial L}{\partial V_{t-1}}$

返回类型 tuple

hard_reset_forward_with_grad_template 和*soft_reset_forward_with_grad_template* 的反向传播。梯度的计算按照

$$\begin{aligned}
\frac{\partial L}{\partial H_t} &= \frac{\partial L}{\partial S_t}\frac{\partial S_t}{\partial H_t} + \frac{\partial L}{\partial V_t}\frac{\partial V_t}{\partial H_t} \\
\frac{\partial L}{\partial X_t} &= \frac{\partial L}{\partial H_t}\frac{\partial H_t}{\partial X_t} \\
\frac{\partial L}{\partial V_{t-1}} &= \frac{\partial L}{\partial H_t}\frac{\partial H_t}{\partial V_{t-1}}
\end{aligned}$$

- *中文 API*

参数

- **grad_spike** (*torch.Tensor*) –$\frac{\partial L}{\partial S_t}$

- **grad_v_next** (torch.Tensor) – $\frac{\partial L}{\partial V_t}$
- **grad_s_to_h** (torch.Tensor) – $\frac{\partial S_t}{\partial H_t}$
- **grad_v_to_h** (torch.Tensor) – $\frac{\partial V_t}{\partial H_t}$

返回 (grad_x, grad_v), where grad_x is $\frac{\partial L}{\partial X_t}$, grad_v is $\frac{\partial L}{\partial V_{t-1}}$

返回类型 tuple

The backward of *hard_reset_forward_with_grad_template* and *soft_reset_forward_with_grad_template*. The gradients are calculated by

$$\begin{aligned}\frac{\partial L}{\partial H_t} &= \frac{\partial L}{\partial S_t}\frac{\partial S_t}{\partial H_t} + \frac{\partial L}{\partial V_t}\frac{\partial V_t}{\partial H_t}\\ \frac{\partial L}{\partial X_t} &= \frac{\partial L}{\partial H_t}\frac{\partial H_t}{\partial X_t}\\ \frac{\partial L}{\partial V_{t-1}} &= \frac{\partial L}{\partial H_t}\frac{\partial H_t}{\partial V_{t-1}}\end{aligned}$$

spikingjelly.cext.neuron.**bptt_template**(*grad_spike_seq: torch.Tensor, grad_v_next: torch.Tensor, grad_s_to_h: torch.Tensor, grad_v_to_h: torch.Tensor, *args, **kwargs*)

- *API in English*

参数

- **grad_spike_seq** (torch.Tensor) – $\frac{\partial L}{\partial S_t}, t = 0, 1, ..., T-1$
- **grad_v_next** (torch.Tensor) – $\frac{\partial L}{\partial V_{T-1}}$
- **grad_s_to_h** (torch.Tensor) – $\frac{\partial S_t}{\partial H_t}, t = 0, 1, ..., T-1$
- **grad_v_to_h** (torch.Tensor) – $\frac{\partial V_t}{\partial H_t}, t = 0, 1, ..., T-1$

返回 (grad_x_seq, grad_v)，其中 grad_x_seq 是 $\frac{\partial L}{\partial X_t}, t = 0, 1, ..., T-1$，grad_v 是 $\frac{\partial L}{\partial V_{-1}}$

返回类型 tuple

backward_template 的多步版本。

- *中文 API*

参数

- **grad_spike_seq** (torch.Tensor) – $\frac{\partial L}{\partial S_t}, t = 0, 1, ..., T-1$
- **grad_v_next** (torch.Tensor) – $\frac{\partial L}{\partial V_{T-1}}$
- **grad_s_to_h** (torch.Tensor) – $\frac{\partial S_t}{\partial H_t}, t = 0, 1, ..., T-1$
- **grad_v_to_h** (torch.Tensor) – $\frac{\partial V_t}{\partial H_t}, t = 0, 1, ..., T-1$

返回 (grad_x_seq, grad_v), where grad_x_seq is $\frac{\partial L}{\partial X_t}, t = 0, 1, ..., T-1$, grad_v is $\frac{\partial L}{\partial V_{-1}}$

返回类型 tuple

The multi-step version of *backward_template*.

spikingjelly.cext.neuron.**LIF_hard_reset_forward**(*x: torch.Tensor, v: torch.Tensor, v_threshold: float, v_reset: float, reciprocal_tau: float, detach_input: bool*)

- *API in English*

参数 **reciprocal_tau** (*float*) – $\frac{1}{\tau}$

其余的参数参见*hard_reset_forward_template*。

对 LIF 神经元进行单步的电压更新，其中电压重置方式是硬重置 (hard reset)。充电的方程为

$$H_t = V_{t-1} + \frac{1}{\tau}(X_t - (V_{t-1} - V_{reset}))$$

- *中文 API*

参数 **reciprocal_tau** (*float*) – $\frac{1}{\tau}$

See *hard_reset_forward_template* for more details about other args。

Update the membrane potential of the LIF neuron by one time step with hard reset. The charging equation is

$$H_t = V_{t-1} + \frac{1}{\tau}(X_t - (V_{t-1} - V_{reset}))$$

spikingjelly.cext.neuron.**LIF_hard_reset_fptt**(*x_seq: torch.Tensor, v: torch.Tensor, v_threshold: float, v_reset: float, reciprocal_tau: float, detach_input: bool*)

- *API in English*

参数 **reciprocal_tau** (*float*) – $\frac{1}{\tau}$

其余的参数参见*hard_reset_fptt_template*。

LIF_hard_reset_forward 的多步版本。

- *中文 API*

参数 **reciprocal_tau** (*float*) – $\frac{1}{\tau}$

See *hard_reset_fptt_template* for more details about other args。

The multi-step version of *LIF_hard_reset_forward*.

spikingjelly.cext.neuron.**LIF_hard_reset_forward_with_grad**(*x: torch.Tensor, v: torch.Tensor, v_threshold: float, v_reset: float, alpha: float, detach_reset: bool, grad_surrogate_function_index: int, reciprocal_tau: float, detach_input: bool*)

- *API in English*

参数 **reciprocal_tau**(*float*) – $\frac{1}{\tau}$

其余的参数参见*hard_reset_forward_with_grad_template*。

对 LIF 神经元进行单步的电压更新并计算反向传播所需的梯度，其中电压重置方式是硬重置 (hard reset)。充电的方程为

$$H_t = V_{t-1} + \frac{1}{\tau}(X_t - (V_{t-1} - V_{reset}))$$

- *中文 API*

参数 **reciprocal_tau**(*float*) – $\frac{1}{\tau}$

See *hard_reset_forward_with_grad_template* for more details about other args。

Update the membrane potential of the LIF neuron by one time step with hard reset and calculate the gradients that the backward function needs. The charging equation is

$$H_t = V_{t-1} + \frac{1}{\tau}(X_t - (V_{t-1} - V_{reset}))$$

spikingjelly.cext.neuron.**LIF_hard_reset_fptt_with_grad**(*x_seq: torch.Tensor, v: torch.Tensor, v_threshold: float, v_reset: float, alpha: float, detach_reset: bool, grad_surrogate_function_index: int, reciprocal_tau: float, detach_input: bool*)

- *API in English*

参数 **reciprocal_tau**(*float*) – $\frac{1}{\tau}$

其余的参数参见*hard_reset_fptt_with_grad_template*。

LIF_hard_reset_forward_with_grad 的多步版本。

- *中文 API*

参数 **reciprocal_tau** (*float*) – $\frac{1}{\tau}$

See *hard_reset_fptt_with_grad_template* for more details about other args。

The multi-step version of *LIF_hard_reset_forward_with_grad*.

spikingjelly.cext.neuron.**LIF_backward**(*grad_spike: torch.Tensor, grad_v_next: torch.Tensor, grad_s_to_h: torch.Tensor, grad_v_to_h: float, reciprocal_tau: float, detach_input: bool*)

- *API in English*

参数 **reciprocal_tau** (*float*) – $\frac{1}{\tau}$

其余的参数参见*backward_template*。

梯度的计算按照

$$\frac{\partial H_t}{\partial X_t} = \frac{1}{\tau}$$
$$\frac{\partial H_t}{\partial V_{t-1}} = 1 - \frac{1}{\tau}$$

- *中文 API*

参数 **reciprocal_tau** (*float*) – $\frac{1}{\tau}$

See *backward_template* for more details about other args。

The gradients are calculated by

$$\frac{\partial H_t}{\partial X_t} = \frac{1}{\tau}$$
$$\frac{\partial H_t}{\partial V_{t-1}} = 1 - \frac{1}{\tau}$$

spikingjelly.cext.neuron.**LIF_bptt**(*grad_spike: torch.Tensor, grad_v_next: torch.Tensor, grad_s_to_h: torch.Tensor, grad_v_to_h: float, reciprocal_tau: float, detach_input: bool*)

- *API in English*

参数 **reciprocal_tau** (*float*) – $\frac{1}{\tau}$

其余的参数参见*bptt_template*。

LIF_backward 的多步版本。

梯度的计算按照

$$\frac{\partial H_t}{\partial X_t} = \frac{1}{\tau}$$
$$\frac{\partial H_t}{\partial V_{t-1}} = 1 - \frac{1}{\tau}$$

- 中文 API

参数 **reciprocal_tau** (*float*) – $\frac{1}{\tau}$

See *bptt_template* for more details about other args。

The multi-step version of *LIF_backward*.

The gradients are calculated by

$$\frac{\partial H_t}{\partial X_t} = \frac{1}{\tau}$$
$$\frac{\partial H_t}{\partial V_{t-1}} = 1 - \frac{1}{\tau}$$

class spikingjelly.cext.neuron.**LIFStep**

基类: torch.autograd.function.Function

static forward(*ctx*, *x*, *v*, *v_threshold*, *v_reset*, *alpha*, *detach_reset*, *grad_surrogate_function_index*, *reciprocal_tau*, *detach_input*)

static backward(*ctx*, *grad_spike*, *grad_v_next*)

class spikingjelly.cext.neuron.**LIFMultiStep**

基类: torch.autograd.function.Function

static forward(*ctx*, *x_seq*, *v*, *v_threshold*, *v_reset*, *alpha*, *detach_reset*, *grad_surrogate_function_index*, *reciprocal_tau*, *detach_input*)

static backward(*ctx*, *grad_spike_seq*, *grad_v_next*)

class spikingjelly.cext.neuron.**IFStep**

基类: torch.autograd.function.Function

static forward(*ctx*, *x*, *v*, *v_threshold*, *v_reset*, *alpha*, *detach_reset*, *grad_surrogate_function_index*)

static backward(*ctx*, *grad_spike*, *grad_v_next*)

class spikingjelly.cext.neuron.**IFMultiStep**

基类: torch.autograd.function.Function

static forward(*ctx*, *x_seq*, *v*, *v_threshold*, *v_reset*, *alpha*, *detach_reset*, *grad_surrogate_function_index*)

static backward(*ctx*, *grad_spike_seq*, *grad_v_next*)

class spikingjelly.cext.neuron.**BaseNode**(*v_threshold=1.0*, *v_reset=0.0*, *surrogate_function='ATan'*, *alpha=2.0*, *detach_reset=False*)

基类: torch.nn.modules.module.Module

reset()

extra_repr()

training: **bool**

class spikingjelly.cext.neuron.**LIFNode**(*tau=100.0, detach_input=False, v_threshold=1.0, v_reset=0.0, surrogate_function='ATan', alpha=2.0, detach_reset=False*)

基类：spikingjelly.cext.neuron.BaseNode

forward(*dv: torch.Tensor*)

extra_repr()

training: bool

class spikingjelly.cext.neuron.**MultiStepLIFNode**(*tau=100.0, detach_input=False, v_threshold=1.0, v_reset=0.0, surrogate_function='ATan', alpha=2.0, detach_reset=False*)

基类：spikingjelly.cext.neuron.LIFNode

forward(*dv_seq: torch.Tensor*)

training: bool

class spikingjelly.cext.neuron.**IFNode**(*v_threshold=1.0, v_reset=0.0, surrogate_function='ATan', alpha=2.0, detach_reset=False*)

基类：spikingjelly.cext.neuron.BaseNode

forward(*dv: torch.Tensor*)

training: bool

class spikingjelly.cext.neuron.**MultiStepIFNode**(*v_threshold=1.0, v_reset=0.0, surrogate_function='ATan', alpha=2.0, detach_reset=False*)

基类：spikingjelly.cext.neuron.IFNode

forward(*dv_seq: torch.Tensor*)

training: bool

Module contents

spikingjelly.cext.**cal_fun_t**(*n, device, f, *args, **kwargs*)

Python 模块索引

S

spikingjelly.cext, 331
spikingjelly.cext.functional, 311
spikingjelly.cext.layer, 313
spikingjelly.cext.neuron, 316
spikingjelly.clock_driven, 284
spikingjelly.clock_driven.encoding, 209
spikingjelly.clock_driven.examples, 209
spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation, 207
spikingjelly.clock_driven.examples.conv_fashion_mnist, 204
spikingjelly.clock_driven.examples.lif_fc_mnist, 204
spikingjelly.clock_driven.functional, 215
spikingjelly.clock_driven.layer, 224
spikingjelly.clock_driven.monitor, 247
spikingjelly.clock_driven.neuron, 242
spikingjelly.clock_driven.rnn, 250
spikingjelly.clock_driven.surrogate, 263
spikingjelly.datasets, 294
spikingjelly.datasets.asl_dvs, 284
spikingjelly.datasets.cifar10_dvs, 285
spikingjelly.datasets.dvs128_gesture, 286
spikingjelly.datasets.n_mnist, 287
spikingjelly.datasets.nav_gesture, 288
spikingjelly.datasets.utils, 289
spikingjelly.event_driven, 303
spikingjelly.event_driven.encoding, 301
spikingjelly.event_driven.examples, 301
spikingjelly.event_driven.examples.tempotron_mnist, 300
spikingjelly.event_driven.neuron, 302
spikingjelly.visualizing, 303

索引

A

ASLDVS (*spikingjelly.datasets* 中的类), 294
ASLDVS (*spikingjelly.datasets.asl_dvs* 中的类), 284
ATan (*spikingjelly.clock_driven.surrogate* 中的类), 273
atan (*spikingjelly.clock_driven.surrogate* 中的类), 273
AutoSparseLinear (*spikingjelly.cext.layer* 中的类), 314
AXAT (*spikingjelly.clock_driven.layer* 中的类), 226

B

backward() (*spikingjelly.cext.functional.sparse_mm_dense_atf* 静态方法), 311
backward() (*spikingjelly.cext.neuron.IFMultiStep* 静态方法), 330
backward() (*spikingjelly.cext.neuron.IFStep* 静态方法), 330
backward() (*spikingjelly.cext.neuron.LIFMultiStep* 静态方法), 330
backward() (*spikingjelly.cext.neuron.LIFStep* 静态方法), 330
backward() (*spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation.relu* 静态方法), 207
backward() (*spikingjelly.clock_driven.surrogate.atan* 静态方法), 273
backward() (*spikingjelly.clock_driven.surrogate.erf* 静态方法), 278
backward() (*spikingjelly.clock_driven.surrogate.nonzero_sign_log_abs* 静态方法), 275
backward() (*spikingjelly.clock_driven.surrogate.piecewise_exp* 静态方法), 267
backward() (*spikingjelly.clock_driven.surrogate.piecewise_leaky_relu* 静态方法), 280
backward() (*spikingjelly.clock_driven.surrogate.piecewise_quadratic* 静态方法), 264
backward() (*spikingjelly.clock_driven.surrogate.sigmoid* 静态方法), 269
backward() (*spikingjelly.clock_driven.surrogate.soft_sign* 静态方法), 271
backward_template() (在 *spikingjelly.cext.neuron* 模块中), 325
base_cell() (*spikingjelly.clock_driven.rnn.SpikingGRU* 静态方法), 262
base_cell() (*spikingjelly.clock_driven.rnn.SpikingLSTM* 静态方法), 261
base_cell() (*spikingjelly.clock_driven.rnn.SpikingRNNBase* 静态方法), 254
base_cell() (*spikingjelly.clock_driven.rnn.SpikingVanillaRNN* 静态方法), 262

BaseEncoder (*spikingjelly.clock_driven.encoding* 中的类), 209
BaseNode (*spikingjelly.cext.neuron* 中的类), 330
BaseNode (*spikingjelly.clock_driven.examples.cifar10_r11_enabling_spike-based_backpropagation* 中的类), 207
BaseNode (*spikingjelly.clock_driven.neuron* 中的类), 242
benchmark() (*spikingjelly.cext.layer.AutoSparseLinear* 方法), 315
bias_hh() (*spikingjelly.clock_driven.rnn.SpikingRNNCellBase* 方法), 252
bias_ih() (*spikingjelly.clock_driven.rnn.SpikingRNNCellBase* 方法), 252
bidirectional_rnn_cell_forward() (在 *spikingjelly.clock_driven.rnn* 模块中), 250
bptt_template() (在 *spikingjelly.cext.neuron* 模块中), 326

C

cal_fun_t() (在 *spikingjelly.cext* 模块中), 331
ChannelsPool (*spikingjelly.clock_driven.layer* 中的类), 235
check_manual_grad() (在 *spikingjelly.clock_driven.surrogate* 模块中), 263
CIFAR10DVS (*spikingjelly.datasets* 中的类), 295
CIFAR10DVS (*spikingjelly.datasets.cifar10_dvs* 中的类), 285
convert_aedat_dir_to_npy_dir() (*spikingjelly.datasets.dvs128_gesture.DVS128Gesture* 静态方法), 287
convert_aedat_dir_to_npy_dir() (*spikingjelly.datasets.DVS128Gesture* 静态方法), 297
convert_events_dir_to_frames_dir() (在 *spikingjelly.datasets.utils* 模块中), 292
create_cells() (*spikingjelly.clock_driven.rnn.SpikingRNNBase* 方法), 253
create_frames_dataset() (*spikingjelly.datasets.asl_dvs.ASLDVS* 静态方法), 285
create_frames_dataset() (*spikingjelly.datasets.ASLDVS* 静态方法), 295
create_frames_dataset() (*spikingjelly.datasets.cifar10_dvs.CIFAR10DVS* 静态方法), 286
create_frames_dataset() (*spikingjelly.datasets.CIFAR10DVS* 静态方法), 296
create_frames_dataset() (*spikingjelly.datasets.dvs128_gesture.DVS128Gesture* 静态方法), 287
create_frames_dataset() (*spikingjelly.datasets.DVS128Gesture* 静态方法), 297
create_frames_dataset() (*spikingjelly.datasets.n_mnist.NMNIST* 静态方法), 288
create_frames_dataset() (*spikingjelly.datasets.nav_gesture.NAVGesture* 静态方法), 289
create_frames_dataset() (*spikingjelly.datasets.NAVGesture* 静态方法), 299
create_frames_dataset() (*spikingjelly.datasets.NMNIST* 静态方法), 298
create_frames_dataset() (*spikingjelly.datasets.utils.EventsFramesDatasetBase* 静态方法), 294
create_mask() (*spikingjelly.clock_driven.layer.Dropout* 方法), 228
create_mask() (*spikingjelly.clock_driven.layer.Dropout2d* 方法), 229

D

DCT (*spikingjelly.clock_driven.layer* 中的类), 226
disable() (*spikingjelly.clock_driven.monitor.Monitor* 方法), 248
download_and_extract() (*spikingjelly.datasets.asl_dvs.ASLDVS* 静态方法), 284
download_and_extract() (*spikingjelly.datasets.ASLDVS* 静态方法), 295
download_and_extract() (*spikingjelly.datasets.cifar10_dvs.CIFAR10DVS* 静

态方法), 286
download_and_extract() (*spikingjelly.datasets.CIFAR10DVS* 静态方法), 296
download_and_extract() (*spikingjelly.datasets.dvs128_gesture.DVS128Gesture* 静态方法), 287
download_and_extract() (*spikingjelly.datasets.DVS128Gesture* 静态方法), 297
download_and_extract() (*spikingjelly.datasets.n_mnist.NMNIST* 静态方法), 288
download_and_extract() (*spikingjelly.datasets.nav_gesture.NAVGesture* 静态方法), 289
download_and_extract() (*spikingjelly.datasets.NAVGesture* 静态方法), 299
download_and_extract() (*spikingjelly.datasets.NMNIST* 静态方法), 298
download_and_extract() (*spikingjelly.datasets.utils.EventsFramesDatasetBase* 静态方法), 294
drop() (*spikingjelly.clock_driven.layer.DropConnectLinear* 方法), 238
DropConnectLinear (*spikingjelly.clock_driven.layer* 中的类), 236
Dropout (*spikingjelly.clock_driven.layer* 中的类), 227
Dropout2d (*spikingjelly.clock_driven.layer* 中的类), 229
DVS128Gesture (*spikingjelly.datasets* 中的类), 296
DVS128Gesture (*spikingjelly.datasets.dvs128_gesture* 中的类), 286

E

enable() (*spikingjelly.clock_driven.monitor.Monitor* 方法), 248
encode() (*spikingjelly.event_driven.encoding.GaussianTuning* 方法), 301
Erf (*spikingjelly.clock_driven.surrogate* 中的类), 278
erf (*spikingjelly.clock_driven.surrogate* 中的类), 278
EventsFramesDatasetBase (*spikingjelly.datasets.utils* 中的类), 293
extra_repr() (*spikingjelly.cext.layer.AutoSparseLinear* 方法), 315
extra_repr() (*spikingjelly.cext.neuron.BaseNode* 方法), 330
extra_repr() (*spikingjelly.cext.neuron.LIFNode* 方法), 331
extra_repr() (*spikingjelly.clock_driven.layer.DropConnectLinear* 方法), 238
extra_repr() (*spikingjelly.clock_driven.layer.Dropout* 方法), 228
extra_repr() (*spikingjelly.clock_driven.layer.SynapseFilter* 方法), 235
extra_repr() (*spikingjelly.clock_driven.neuron.BaseNode* 方法), 243
extra_repr() (*spikingjelly.clock_driven.neuron.LIFNode* 方法), 247
extra_repr() (*spikingjelly.clock_driven.surrogate.MultiArgsSurrogateFunctionBase* 方法), 264
extra_repr() (*spikingjelly.clock_driven.surrogate.SurrogateFunctionBase* 方法), 264
extract_zip_in_dir() (在 *spikingjelly.datasets.utils* 模块中), 293

F

first_spike_index() (在 *spikingjelly.clock_driven.functional* 模块中), 222
forward() (*spikingjelly.cext.functional.sparse_mm_dense_atf* 静态方法), 311
forward() (*spikingjelly.cext.layer.AutoSparseLinear* 方法), 315
forward() (*spikingjelly.cext.layer.SparseLinear* 方法), 314
forward() (*spikingjelly.cext.neuron.IFMultiStep* 静态方法), 330
forward() (*spikingjelly.cext.neuron.IFNode* 方法), 331
forward() (*spikingjelly.cext.neuron.IFStep* 静态方法),

330
forward() (*spikingjelly.cext.neuron.LIFMultiStep* 静态方法), 330
forward() (*spikingjelly.cext.neuron.LIFNode* 方法), 331
forward() (*spikingjelly.cext.neuron.LIFStep* 静态方法), 330
forward() (*spikingjelly.cext.neuron.MultiStepIFNode* 方法), 331
forward() (*spikingjelly.cext.neuron.MultiStepLIFNode* 方法), 331
forward() (*spikingjelly.clock_driven.encoding.BaseEncoder* 方法), 210
forward() (*spikingjelly.clock_driven.encoding.GaussianTuningCurveEncoder* 方法), 213
forward() (*spikingjelly.clock_driven.encoding.LatencyEncoder* 方法), 212
forward() (*spikingjelly.clock_driven.encoding.PeriodicEncoder* 方法), 210
forward() (*spikingjelly.clock_driven.encoding.PoissonEncoder* 方法), 212
forward() (*spikingjelly.clock_driven.encoding.WeightedPhaseEncoder* 方法), 215
forward() (*spikingjelly.clock_driven.examples.cifar10_r11* 方法), 208
forward() (*spikingjelly.clock_driven.examples.cifar10_r11* 方法), 209
forward() (*spikingjelly.clock_driven.examples.cifar10_r11* 方法), 209
forward() (*spikingjelly.clock_driven.examples.cifar10_r11* 静态方法), 207
forward() (*spikingjelly.clock_driven.examples.cifar10_r11* 方法), 209
forward() (*spikingjelly.clock_driven.examples.conv_fashion_mnist.Net* 方法), 204
forward() (*spikingjelly.clock_driven.layer.AXAT* 方法), 227
forward() (*spikingjelly.clock_driven.layer.ChannelsPool* 方法), 236
forward() (*spikingjelly.clock_driven.layer.DCT* 方法), 226
forward() (*spikingjelly.clock_driven.layer.DropConnectLinear* 方法), 238
forward() (*spikingjelly.clock_driven.layer.Dropout* 方法), 228
forward() (*spikingjelly.clock_driven.layer.MultiStepContainer* 方法), 238
forward() (*spikingjelly.clock_driven.layer.MultiStepDropout* 方法), 230
forward() (*spikingjelly.clock_driven.layer.MultiStepDropout2d* 方法), 231
forward() (*spikingjelly.clock_driven.layer.NeuNorm* 方法), 225
forward() (*spikingjelly.clock_driven.layer.PrintShapeModule* 方法), 242
forward() (*spikingjelly.clock_driven.layer.SeqToANNContainer* 方法), 239
forward() (*spikingjelly.clock_driven.layer.SynapseFilter* 方法), 235
forward() (*spikingjelly.clock_driven.neuron.BaseNode* 方法), 244
forward() (*spikingjelly.clock_driven.rnn.SpikingGRUCell* 方法), 262
forward() (*spikingjelly.clock_driven.rnn.SpikingLSTMCell* 方法), 258
forward() (*spikingjelly.clock_driven.rnn.BaseSpikingRNNBase* 方法), 255
forward() (*spikingjelly.clock_driven.rnn.SpikingVanillaRNNCell* 方法), 262
forward() (*spikingjelly.clock_driven.surrogate.atan* 静态方法), 273
forward() (*spikingjelly.clock_driven.surrogate.erf* 静态方法), 278
forward() (*spikingjelly.clock_driven.surrogate.nonzero_sign_log_abs* 静态方法), 275
forward() (*spikingjelly.clock_driven.surrogate.piecewise_exp* 静态方法), 266
forward() (*spikingjelly.clock_driven.surrogate.piecewise_leaky_relu* 静态方法), 280
forward() (*spikingjelly.clock_driven.surrogate.piecewise_quadratic* 静态方法), 264
forward() (*spikingjelly.clock_driven.surrogate.PiecewiseLeakyReLU* 方法), 283
forward() (*spikingjelly.clock_driven.surrogate.sigmoid* 静态方法), 269

forward() (*spikingjelly.clock_driven.surrogate.soft_sign* 静态方法), 271
forward() (*spikingjelly.clock_driven.surrogate.SurrogateFunctionBase* 方法), 264
forward() (*spikingjelly.event_driven.examples.tempotron_mnist.Net* 方法), 300
forward() (*spikingjelly.event_driven.neuron.Tempotron* 方法), 302
forward_hook() (*spikingjelly.clock_driven.monitor.Monitor* 方法), 248
FunctionThread (*spikingjelly.datasets.utils* 中的类), 289

G

GaussianTuning (*spikingjelly.event_driven.encoding* 中的类), 301
GaussianTuningCurveEncoder (*spikingjelly.clock_driven.encoding* 中的类), 212
get_avg_firing_rate() (*spikingjelly.clock_driven.monitor.Monitor* 方法), 248
get_events_item() (*spikingjelly.datasets.asl_dvs.ASLDVS* 静态方法), 285
get_events_item() (*spikingjelly.datasets.ASLDVS* 静态方法), 295
get_events_item() (*spikingjelly.datasets.cifar10_dvs.CIFAR10DVS* 静态方法), 286
get_events_item() (*spikingjelly.datasets.CIFAR10DVS* 静态方法), 296
get_events_item() (*spikingjelly.datasets.dvs128_gesture.DVS128Gesture* 静态方法), 287
get_events_item() (*spikingjelly.datasets.DVS128Gesture* 静态方法), 297
get_events_item() (*spikingjelly.datasets.n_mnist.NMNIST* 静态方法), 288
get_events_item() (*spikingjelly.datasets.nav_gesture.NAVGesture* 静态方法), 289
get_events_item() (*spikingjelly.datasets.NAVGesture* 静态方法), 299
get_events_item() (*spikingjelly.datasets.NMNIST* 静态方法), 298
get_events_item() (*spikingjelly.datasets.utils.EventsFramesDatasetBase* 静态方法), 293
get_frames_item() (*spikingjelly.datasets.asl_dvs.ASLDVS* 静态方法), 285
get_frames_item() (*spikingjelly.datasets.ASLDVS* 静态方法), 295
get_frames_item() (*spikingjelly.datasets.cifar10_dvs.CIFAR10DVS* 静态方法), 286
get_frames_item() (*spikingjelly.datasets.CIFAR10DVS* 静态方法), 296
get_frames_item() (*spikingjelly.datasets.dvs128_gesture.DVS128Gesture* 静态方法), 287
get_frames_item() (*spikingjelly.datasets.DVS128Gesture* 静态方法), 297
get_frames_item() (*spikingjelly.datasets.n_mnist.NMNIST* 静态方法), 288
get_frames_item() (*spikingjelly.datasets.nav_gesture.NAVGesture* 静态方法), 289
get_frames_item() (*spikingjelly.datasets.NAVGesture* 静态方法), 299
get_frames_item() (*spikingjelly.datasets.NMNIST* 静态方法), 298
get_frames_item() (*spikingjelly.datasets.utils.EventsFramesDatasetBase* 静态方法), 293
get_label() (*spikingjelly.datasets.nav_gesture.NAVGesture* 静态方法), 289
get_label() (*spikingjelly.datasets.NAVGesture* 静态方

法), 299
get_nonfire_ratio() (*spikingjelly.clock_driven.monitor.Monitor* 方法), 249
get_wh() (*spikingjelly.datasets.asl_dvs.ASLDVS* 静态方法), 284
get_wh() (*spikingjelly.datasets.ASLDVS* 静态方法), 295
get_wh() (*spikingjelly.datasets.cifar10_dvs.CIFAR10DVS* 静态方法), 286
get_wh() (*spikingjelly.datasets.CIFAR10DVS* 静态方法), 296
get_wh() (*spikingjelly.datasets.dvs128_gesture.DVS128Gesture* 静态方法), 287
get_wh() (*spikingjelly.datasets.DVS128Gesture* 静态方法), 297
get_wh() (*spikingjelly.datasets.n_mnist.NMNIST* 静态方法), 288
get_wh() (*spikingjelly.datasets.nav_gesture.NAVGesture* 静态方法), 289
get_wh() (*spikingjelly.datasets.NAVGesture* 静态方法), 298
get_wh() (*spikingjelly.datasets.NMNIST* 静态方法), 297
get_wh() (*spikingjelly.datasets.utils.EventsFramesDatasetBase* 静态方法), 293

H

hard_reset_forward_template() (在 *spikingjelly.cext.neuron* 模块中), 316
hard_reset_forward_with_grad_template() (在 *spikingjelly.cext.neuron* 模块中), 320
hard_reset_fptt_template() (在 *spikingjelly.cext.neuron* 模块中), 317
hard_reset_fptt_with_grad_template() (在 *spikingjelly.cext.neuron* 模块中), 321
heaviside() (在 *spikingjelly.clock_driven.surrogate* 模块中), 263

I

IFMultiStep (*spikingjelly.cext.neuron* 中的类), 330
IFNode (*spikingjelly.cext.neuron* 中的类), 331
IFNode (*spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation* 中的类), 209
IFNode (*spikingjelly.clock_driven.neuron* 中的类), 244
IFStep (*spikingjelly.cext.neuron* 中的类), 330
in_features (*spikingjelly.cext.layer.AutoSparseLinear* 属性), 316
in_features (*spikingjelly.cext.layer.SparseLinear* 属性), 314
integrate_events_to_frames() (在 *spikingjelly.datasets.utils* 模块中), 289
IntervalEncoder (*spikingjelly.clock_driven.encoding* 中的类), 214

K

kernel_dot_product() (在 *spikingjelly.clock_driven.functional* 模块中), 219

L

LatencyEncoder (*spikingjelly.clock_driven.encoding* 中的类), 211
LIF_backward() (在 *spikingjelly.cext.neuron* 模块中), 329
LIF_bptt() (在 *spikingjelly.cext.neuron* 模块中), 329
LIF_hard_reset_forward() (在 *spikingjelly.cext.neuron* 模块中), 327
LIF_hard_reset_forward_with_grad() (在 *spikingjelly.cext.neuron* 模块中), 327
LIF_hard_reset_fptt() (在 *spikingjelly.cext.neuron* 模块中), 327
LIF_hard_reset_fptt_with_grad() (在 *spikingjelly.cext.neuron* 模块中), 328
LIFMultiStep (*spikingjelly.cext.neuron* 中的类), 330
LIFNode (*spikingjelly.cext.neuron* 中的类), 331
LIFNode (*spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebas* 中的类), 208
LIFNode (*spikingjelly.clock_driven.neuron* 中的类), 246
LIFStep (*spikingjelly.cext.neuron* 中的类), 330
load_events() (在 *spikingjelly.datasets.cifar10_dvs* 模块中), 285
load_raw_events() (在 *spikingjelly.datasets.cifar10_dvs* 模块中), 285

M

main() (在 *spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikeb* 模块中), 209

main() (在 spikingjelly.clock_driven.examples.conv_fashion_mnist 模块中), 204
main() (在 spikingjelly.clock_driven.examples.lif_fc_mnist 模块中), 204
main() (在 spikingjelly.event_driven.examples.tempotron_mnist 模块中), 300
Monitor (spikingjelly.clock_driven.monitor 中的类), 247
mse_loss() (spikingjelly.event_driven.neuron.Tempotron 静态方法), 302
MultiArgsSurrogateFunctionBase (spikingjelly.clock_driven.surrogate 中的类), 264
MultiStepContainer (spikingjelly.clock_driven.layer 中的类), 238
MultiStepDropout (spikingjelly.clock_driven.layer 中的类), 229
MultiStepDropout2d (spikingjelly.clock_driven.layer 中的类), 230
MultiStepIFNode (spikingjelly.cext.neuron 中的类), 331
MultiStepLIFNode (spikingjelly.cext.neuron 中的类), 331

N

NAVGesture (spikingjelly.datasets 中的类), 298
NAVGesture (spikingjelly.datasets.nav_gesture 中的类), 288
Net (spikingjelly.clock_driven.examples.conv_fashion_mnist 中的类), 204
Net (spikingjelly.event_driven.examples.tempotron_mnist 中的类), 300
NeuNorm (spikingjelly.clock_driven.layer 中的类), 224
neuronal_charge() (spikingjelly.clock_driven.neuron.BaseNode 方法), 243
neuronal_charge() (spikingjelly.clock_driven.neuron.IFNode 方法), 245
neuronal_charge() (spikingjelly.clock_driven.neuron.LIFNode 方法), 247
neuronal_fire() (spikingjelly.clock_driven.neuron.BaseNode 方法), 243
neuronal_reset() (spikingjelly.clock_driven.neuron.BaseNode 方法), 243
NMNIST (spikingjelly.datasets 中的类), 297
NMNIST (spikingjelly.datasets.n_mnist 中的类), 287
nonzero_sign_log_abs (spikingjelly.clock_driven.surrogate 中的类), 275
NonzeroSignLogAbs (spikingjelly.clock_driven.surrogate 中的类), 275
normalize_frame() (在 spikingjelly.datasets.utils 模块中), 292

O

out_features (spikingjelly.cext.layer.AutoSparseLinear 属性), 316
out_features (spikingjelly.cext.layer.SparseLinear 属性), 314

P

parse_raw_address() (在 spikingjelly.datasets.cifar10_dvs 模块中), 285
PeriodicEncoder (spikingjelly.clock_driven.encoding 中的类), 210
piecewise_exp (spikingjelly.clock_driven.surrogate 中的类), 266
piecewise_leaky_relu (spikingjelly.clock_driven.surrogate 中的类), 280
piecewise_quadratic (spikingjelly.clock_driven.surrogate 中的类), 264
PiecewiseExp (spikingjelly.clock_driven.surrogate 中的类), 267
PiecewiseLeakyReLU (spikingjelly.clock_driven.surrogate 中的类), 280
PiecewiseQuadratic (spikingjelly.clock_driven.surrogate 中的类), 264
plot_1d_spikes() (在 spikingjelly.visualizing 模块中), 307
plot_2d_bar_in_3d() (在 spikingjelly.visualizing 模块中), 304
plot_2d_heatmap() (在 spikingjelly.visualizing 模块中), 303

plot_2d_spiking_feature_map() (在 spikingjelly.visualizing 模块中), 308
plot_one_neuron_v_s() (在 spikingjelly.visualizing 模块中), 310
PoissonEncoder (spikingjelly.clock_driven.encoding 中的类), 212
primitive_function() (spikingjelly.clock_driven.surrogate.ATan 静态方法), 275
primitive_function() (spikingjelly.clock_driven.surrogate.Erf 静态方法), 280
primitive_function() (spikingjelly.clock_driven.surrogate.NonzeroSignLogAbs 静态方法), 278
primitive_function() (spikingjelly.clock_driven.surrogate.PiecewiseExp 静态方法), 269
primitive_function() (spikingjelly.clock_driven.surrogate.PiecewiseLeakyReLU 静态方法), 283
primitive_function() (spikingjelly.clock_driven.surrogate.PiecewiseQuadratic 静态方法), 266
primitive_function() (spikingjelly.clock_driven.surrogate.Sigmoid 静态方法), 271
primitive_function() (spikingjelly.clock_driven.surrogate.SoftSign 静态方法), 273
primitive_function() (spikingjelly.clock_driven.surrogate.SurrogateFunctionBase 静态方法), 264
PrintShapeModule (spikingjelly.clock_driven.layer 中的类), 241
psp_kernel() (spikingjelly.event_driven.neuron.Tempotron 静态方法), 302

R

read_bin() (spikingjelly.datasets.asl_dvs.ASLDVS 静态方法), 285
read_bin() (spikingjelly.datasets.ASLDVS 静态方法), 295
read_bin() (spikingjelly.datasets.cifar10_dvs.CIFAR10DVS 静态方法), 286
read_bin() (spikingjelly.datasets.CIFAR10DVS 静态方法), 296
read_bin() (spikingjelly.datasets.dvs128_gesture.DVS128Gesture 静态方法), 287
read_bin() (spikingjelly.datasets.DVS128Gesture 静态方法), 297
read_bin() (spikingjelly.datasets.n_mnist.NMNIST 静态方法), 288
read_bin() (spikingjelly.datasets.nav_gesture.NAVGesture 静态方法), 289
read_bin() (spikingjelly.datasets.NAVGesture 静态方法), 298
read_bin() (spikingjelly.datasets.NMNIST 静态方法), 297
read_bin() (spikingjelly.datasets.utils.EventsFramesDatasetBase 静态方法), 293
read_bits() (在 spikingjelly.datasets.cifar10_dvs 模块中), 285
redundant_one_hot() (在 spikingjelly.clock_driven.functional 模块中), 221
relu (spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_l 中的类), 207
reset() (spikingjelly.cext.neuron.BaseNode 方法), 330
reset() (spikingjelly.clock_driven.encoding.BaseEncoder 方法), 210
reset() (spikingjelly.clock_driven.encoding.GaussianTuningCurveEncoder 方法), 214
reset() (spikingjelly.clock_driven.encoding.IntervalEncoder 方法), 214
reset() (spikingjelly.clock_driven.encoding.LatencyEncoder 方法), 212
reset() (spikingjelly.clock_driven.encoding.PeriodicEncoder 方法), 211
reset() (spikingjelly.clock_driven.encoding.WeightedPhaseEncoder

方法), 215
reset() (spikingjelly.clock_driven.examples.cifar10_r11_enabling_spike_based_backpropagation.ResNet11 方法), 208
reset() (spikingjelly.clock_driven.layer.DropConnectLinear 方法), 238
reset() (spikingjelly.clock_driven.layer.Dropout 方法), 228
reset() (spikingjelly.clock_driven.layer.MultiStepContainer 方法), 238
reset() (spikingjelly.clock_driven.layer.NeuNorm 方法), 225
reset() (spikingjelly.clock_driven.layer.STDPLearner 方法), 241
reset() (spikingjelly.clock_driven.layer.SynapseFilter 方法), 235
reset() (spikingjelly.clock_driven.monitor.Monitor 方法), 248
reset() (spikingjelly.clock_driven.neuron.BaseNode 方法), 244
reset_() (spikingjelly.clock_driven.examples.cifar10_r11_enabling_spike_based_backpropagation.ResNet11 方法), 209
reset_net() (在 spikingjelly.clock_driven.functional 模块中), 215
reset_parameters() (spikingjelly.clock_driven.layer.DropConnectLinear 方法), 237
reset_parameters() (spikingjelly.clock_driven.rnn.SpikingRNNCellBase 方法), 251
ResNet11 (spikingjelly.clock_driven.examples.cifar10_r11_enabling_spike_based_backpropagation 中的类), 209
run() (spikingjelly.datasets.utils.FunctionThread 方法), 289

S

SeqToANNContainer (spikingjelly.clock_driven.layer 中的类), 238
set_monitor() (spikingjelly.clock_driven.neuron.BaseNode 方法), 243
set_monitor() (在 spikingjelly.clock_driven.functional 模块中), 215
set_out_spike() (spikingjelly.clock_driven.encoding.PeriodicEncoder 方法), 210
set_spiking_mode() (spikingjelly.clock_driven.surrogate.MultiArgsSurrogateFunctionBase 方法), 264
set_spiking_mode() (spikingjelly.clock_driven.surrogate.SurrogateFunctionBase 方法), 264
set_threshold_margin() (在 spikingjelly.clock_driven.functional 模块中), 220
Sigmoid (spikingjelly.clock_driven.surrogate 中的类), 269
sigmoid (spikingjelly.clock_driven.surrogate 中的类), 269
skip_header() (在 spikingjelly.datasets.cifar10_dvs 模块中), 285
soft_reset_forward_template() (在 spikingjelly.cext.neuron 模块中), 318
soft_reset_forward_with_grad_template() (在 spikingjelly.cext.neuron 模块中), 322
soft_reset_fptt_template() (在 spikingjelly.cext.neuron 模块中), 319
soft_reset_fptt_with_grad_template() (在 spikingjelly.cext.neuron 模块中), 324
soft_sign (spikingjelly.clock_driven.surrogate 中的类), 271
SoftSign (spikingjelly.clock_driven.surrogate 中的类), 271
sparse_mm_dense_backpropagation (在 spikingjelly.cext.functional 模块中), 311
sparse_mm_dense_atf (spikingjelly.cext.functional 中的类), 311
SparseLinear (spikingjelly.cext.layer 中的类), 313
SPEECHCOMMANDS (spikingjelly.datasets 中的类), 299
spike_cluster() (在 spikingjelly.clock_driven.functional 模块中), 216
spike_mse_loss() (在 spikingjelly.clock_driven.functional 模块中), 223
spike_similar_loss() (在 spikingjelly.clock_driven.functional 模块中), 217
spiking() (spikingjelly.clock_driven.examples.cifar10_r11_enabling_spike

方法), 208
spiking_function() (*spikingjelly.clock_driven.surrogate.ATan* 静态方法), 275
spiking_function() (*spikingjelly.clock_driven.surrogate.Erf* 静态方法), 280
spiking_function() (*spikingjelly.clock_driven.surrogate.NonzeroSignLogAbs* 静态方法), 277
spiking_function() (*spikingjelly.clock_driven.surrogate.PiecewiseExp* 静态方法), 268
spiking_function() (*spikingjelly.clock_driven.surrogate.PiecewiseLeakyReLU* 静态方法), 283
spiking_function() (*spikingjelly.clock_driven.surrogate.PiecewiseQuadratic* 静态方法), 266
spiking_function() (*spikingjelly.clock_driven.surrogate.Sigmoid* 静态方法), 271
spiking_function() (*spikingjelly.clock_driven.surrogate.SoftSign* 静态方法), 273
spiking_function() (*spikingjelly.clock_driven.surrogate.SurrogateFunctionBase* 静态方法), 264
SpikingGRU (*spikingjelly.clock_driven.rnn* 中的类), 262
SpikingGRUCell (*spikingjelly.clock_driven.rnn* 中的类), 262
spikingjelly.cext
模块, 331
spikingjelly.cext.functional
模块, 311
spikingjelly.cext.layer
模块, 313
spikingjelly.cext.neuron
模块, 316
spikingjelly.clock_driven
模块, 284
spikingjelly.clock_driven.encoding
模块, 209
spikingjelly.clock_driven.examples
模块, 209
spikingjelly.clock_driven.examples.cifar10_r11_enab
模块, 207
spikingjelly.clock_driven.examples.conv_fashion_mni
模块, 204
spikingjelly.clock_driven.examples.lif_fc_mnist
模块, 204
spikingjelly.clock_driven.functional
模块, 215
spikingjelly.clock_driven.layer
模块, 224
spikingjelly.clock_driven.monitor
模块, 247
spikingjelly.clock_driven.neuron
模块, 242
spikingjelly.clock_driven.rnn
模块, 250
spikingjelly.clock_driven.surrogate
模块, 263
spikingjelly.datasets
模块, 294
spikingjelly.datasets.asl_dvs
模块, 284
spikingjelly.datasets.cifar10_dvs
模块, 285
spikingjelly.datasets.dvs128_gesture
模块, 286
spikingjelly.datasets.n_mnist
模块, 287
spikingjelly.datasets.nav_gesture
模块, 288
spikingjelly.datasets.utils
模块, 289
spikingjelly.event_driven
模块, 303
spikingjelly.event_driven.encoding
模块, 301
spikingjelly.event_driven.examples
模块, 301

spikingjelly.event_driven.examples.tempotron_mnist 模块, 300
spikingjelly.event_driven.neuron 模块, 302
spikingjelly.visualizing 模块, 303
SpikingLSTM (*spikingjelly.clock_driven.rnn* 中的类), 259
SpikingLSTMCell (*spikingjelly.clock_driven.rnn* 中的类), 256
SpikingRNNBase (*spikingjelly.clock_driven.rnn* 中的类), 252
SpikingRNNCellBase (*spikingjelly.clock_driven.rnn* 中的类), 250
SpikingVanillaRNN (*spikingjelly.clock_driven.rnn* 中的类), 262
SpikingVanillaRNNCell (*spikingjelly.clock_driven.rnn* 中的类), 261
states_num() (*spikingjelly.clock_driven.rnn.SpikingGRU* 静态方法), 262
states_num() (*spikingjelly.clock_driven.rnn.SpikingLSTM* 静态方法), 261
states_num() (*spikingjelly.clock_driven.rnn.SpikingRNNBase* 静态方法), 254
states_num() (*spikingjelly.clock_driven.rnn.SpikingVanillaRNN* 静态方法), 262
stdp() (*spikingjelly.clock_driven.layer.STDPLearner* 方法), 241
STDPLearner (*spikingjelly.clock_driven.layer* 中的类), 239
step() (*spikingjelly.clock_driven.encoding.BaseEncoder* 方法), 210
step() (*spikingjelly.clock_driven.encoding.GaussianTuningCurveEncoder* 方法), 214
step() (*spikingjelly.clock_driven.encoding.IntervalEncoder* 方法), 214
step() (*spikingjelly.clock_driven.encoding.LatencyEncoder* 方法), 212
step() (*spikingjelly.clock_driven.encoding.PeriodicEncoder* 方法), 210
step() (*spikingjelly.clock_driven.encoding.WeightedPhaseEncoder* 方法), 215
SurrogateFunctionBase (*spikingjelly.clock_driven.surrogate* 中的类), 263
SynapseFilter (*spikingjelly.clock_driven.layer* 中的类), 231

T

Tempotron (*spikingjelly.event_driven.neuron* 中的类), 302
training (*spikingjelly.cext.neuron.BaseNode* 属性), 330
training (*spikingjelly.cext.neuron.IFNode* 属性), 331
training (*spikingjelly.cext.neuron.LIFNode* 属性), 331
training (*spikingjelly.cext.neuron.MultiStepIFNode* 属性), 331
training (*spikingjelly.cext.neuron.MultiStepLIFNode* 属性), 331
training (*spikingjelly.clock_driven.encoding.BaseEncoder* 属性), 210
training (*spikingjelly.clock_driven.encoding.GaussianTuningCurveEncode* 属性), 214
training (*spikingjelly.clock_driven.encoding.IntervalEncoder* 属性), 214
training (*spikingjelly.clock_driven.encoding.LatencyEncoder* 属性), 212
training (*spikingjelly.clock_driven.encoding.PeriodicEncoder* 属性), 211
training (*spikingjelly.clock_driven.encoding.PoissonEncoder* 属性), 212
training (*spikingjelly.clock_driven.encoding.WeightedPhaseEncoder* 属性), 215
training (*spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikeb* 属性), 208
training (*spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikeb* 属性), 209
training (*spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikeb* 属性), 209
training (*spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikeb* 属性), 209
training (*spikingjelly.clock_driven.examples.conv_fashion_mnist.Net*

属性), 204
training (*spikingjelly.clock_driven.layer.AXAT* 属性), 227
training (*spikingjelly.clock_driven.layer.ChannelsPool* 属性), 236
training (*spikingjelly.clock_driven.layer.DCT* 属性), 226
training (*spikingjelly.clock_driven.layer.DropConnectLinear* 属性), 238
training (*spikingjelly.clock_driven.layer.Dropout* 属性), 229
training (*spikingjelly.clock_driven.layer.Dropout2d* 属性), 229
training (*spikingjelly.clock_driven.layer.MultiStepContainer* 属性), 238
training (*spikingjelly.clock_driven.layer.MultiStepDropout* 属性), 230
training (*spikingjelly.clock_driven.layer.MultiStepDropout2d* 属性), 231
training (*spikingjelly.clock_driven.layer.NeuNorm* 属性), 226
training (*spikingjelly.clock_driven.layer.PrintShapeModule* 属性), 242
training (*spikingjelly.clock_driven.layer.SeqToANNContainer* 属性), 239
training (*spikingjelly.clock_driven.layer.STDPLearner* 属性), 241
training (*spikingjelly.clock_driven.layer.SynapseFilter* 属性), 235
training (*spikingjelly.clock_driven.neuron.BaseNode* 属性), 244
training (*spikingjelly.clock_driven.neuron.IFNode* 属性), 245
training (*spikingjelly.clock_driven.neuron.LIFNode* 属性), 247
training (*spikingjelly.clock_driven.rnn.SpikingGRU* 属性), 262
training (*spikingjelly.clock_driven.rnn.SpikingGRUCell* 属性), 262
training (*spikingjelly.clock_driven.rnn.SpikingLSTM* 属性), 261
training (*spikingjelly.clock_driven.rnn.SpikingLSTMCell* 属性), 259
training (*spikingjelly.clock_driven.rnn.SpikingRNNBase* 属性), 256
training (*spikingjelly.clock_driven.rnn.SpikingRNNCellBase* 属性), 252
training (*spikingjelly.clock_driven.rnn.SpikingVanillaRNN* 属性), 262
training (*spikingjelly.clock_driven.rnn.SpikingVanillaRNNCell* 属性), 262
training (*spikingjelly.clock_driven.surrogate.ATan* 属性), 275
training (*spikingjelly.clock_driven.surrogate.Erf* 属性), 280
training (*spikingjelly.clock_driven.surrogate.MultiArgsSurrogateFunctionB* 属性), 264
training (*spikingjelly.clock_driven.surrogate.NonzeroSignLogAbs* 属性), 278
training (*spikingjelly.clock_driven.surrogate.PiecewiseExp* 属性), 269
training (*spikingjelly.clock_driven.surrogate.PiecewiseLeakyReLU* 属性), 283
training (*spikingjelly.clock_driven.surrogate.PiecewiseQuadratic* 属性), 266
training (*spikingjelly.clock_driven.surrogate.Sigmoid* 属性), 271
training (*spikingjelly.clock_driven.surrogate.SoftSign* 属性), 273
training (*spikingjelly.clock_driven.surrogate.SurrogateFunctionBase* 属性), 264
training (*spikingjelly.event_driven.examples.tempotron_mnist.Net* 属性), 300
training (*spikingjelly.event_driven.neuron.Tempotron* 属性), 303

W

weight (*spikingjelly.cext.layer.AutoSparseLinear* 属性), 316
weight (*spikingjelly.cext.layer.SparseLinear* 属性), 314
weight_hh() (*spikingjelly.clock_driven.rnn.SpikingRNNCellBase* 方法), 251
weight_ih() (*spiking-*

jelly.clock_driven.rnn.SpikingRNNCellBase 方法), 251
WeightedPhaseEncoder (*spikingjelly.clock_driven.encoding* 中的类), 214

模块

spikingjelly.cext, 331
spikingjelly.cext.functional, 311
spikingjelly.cext.layer, 313
spikingjelly.cext.neuron, 316
spikingjelly.clock_driven, 284
spikingjelly.clock_driven.encoding, 209
spikingjelly.clock_driven.examples, 209
spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation, 207
spikingjelly.clock_driven.examples.conv_fashion_mnist, 204
spikingjelly.clock_driven.examples.lif_fc_mnist, 204
spikingjelly.clock_driven.functional, 215
spikingjelly.clock_driven.layer, 224
spikingjelly.clock_driven.monitor, 247
spikingjelly.clock_driven.neuron, 242
spikingjelly.clock_driven.rnn, 250
spikingjelly.clock_driven.surrogate, 263
spikingjelly.datasets, 294
spikingjelly.datasets.asl_dvs, 284
spikingjelly.datasets.cifar10_dvs, 285
spikingjelly.datasets.dvs128_gesture, 286
spikingjelly.datasets.n_mnist, 287
spikingjelly.datasets.nav_gesture, 288
spikingjelly.datasets.utils, 289
spikingjelly.event_driven, 303
spikingjelly.event_driven.encoding, 301
spikingjelly.event_driven.examples, 301
spikingjelly.event_driven.examples.tempotron_mn 300
spikingjelly.event_driven.neuron, 302
spikingjelly.visualizing, 303