spikingjelly

发行版本 *alpha*

PKU MLG, PCL

2022 年 11 月 16 日

上手教程

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。

奇数版本是开发版，随着 GitHub/OpenI 不断更新。偶数版本是稳定版，可以从 PyPI 获取。

从 PyPI 安装最新的稳定版本：

```
pip install spikingjelly
```

从源代码安装最新的开发版：

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

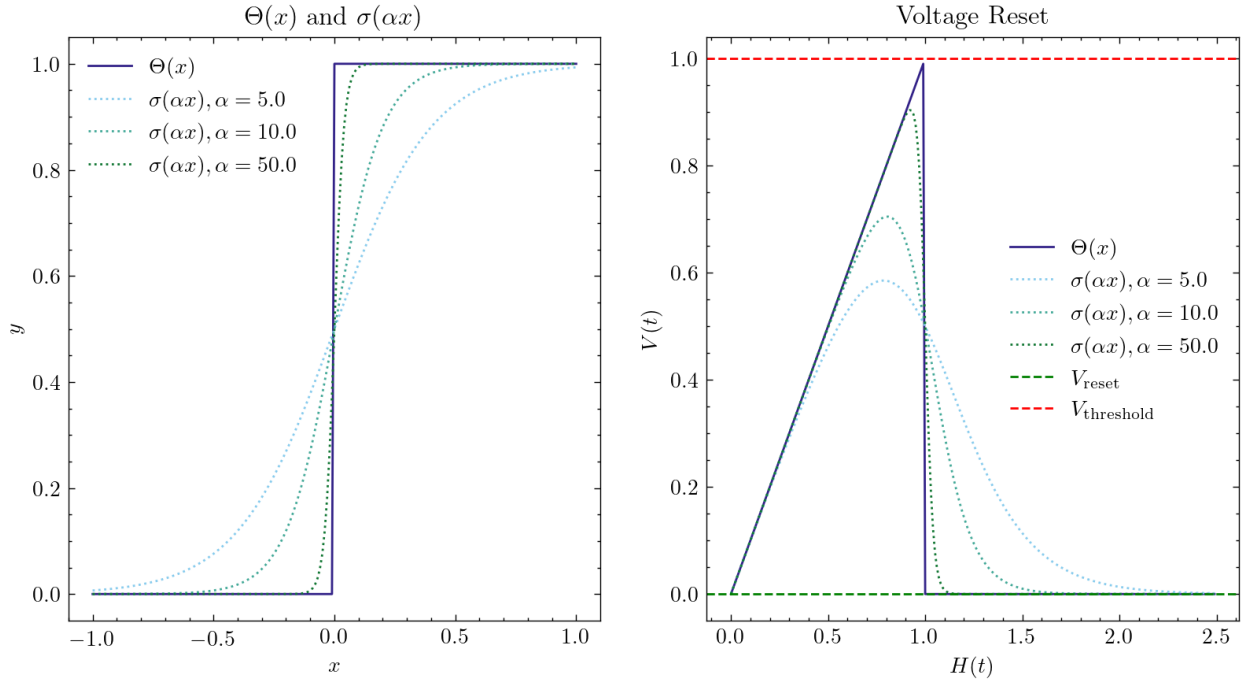
$$
\Theta(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}
$$

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(base.MemoryModule):
    def __init__(self, v_threshold: float = 1., v_reset: float = 0.,
                 surrogate_function: Callable = surrogate.Sigmoid(), detach_reset:␣
↪bool = False):
        """
        :param v_threshold: 神经元的阈值电压
        :type v_threshold: float

        :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 ``v_threshold``
```

(续下页)

(接上页)

```
        :type v_reset: float

        :param surrogate_function: 反向传播时用来计算脉冲函数梯度的替代函数
        :type surrogate_function: Callable

        :param detach_reset: 是否将reset过程的计算图分离
        :type detach_reset: bool

        可微分SNN神经元的基类神经元。
        """
```

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 示例：使用单层全连接网络进行 MNIST 分类

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

首先确定所需超参数：

```
parser.add_argument('--device', default='cuda:0', help=
↪'运行的设备，例如“cpu”或“cuda:0”\n Device, e.g., "cpu" or "cuda:0"')

parser.add_argument('--dataset-dir', default='./', help=
↪'保存MNIST数据集的位置，例如“./”\n Root directory for saving MNIST dataset, e.g.,
↪"./"')
parser.add_argument('--log-dir', default='./', help=
↪'保存tensorboard日志文件的位置，例如“./”\n Root directory for saving tensorboard␣
↪logs, e.g., "./"')
parser.add_argument('--model-output-dir', default='./', help='模型保存路径，例如“./
```

(续下页)

(接上页)

```
↪" \n Model directory for saving, e.g., "./"')

parser.add_argument('-b', '--batch-size', default=64, type=int, help='Batch␣
↪大小，例如“64” \n Batch size, e.g., "64"')
parser.add_argument('-T', '--timesteps', default=100, type=int, dest='T', help=
↪'仿真时长，例如“100” \n Simulating timesteps, e.g., "100"')
parser.add_argument('--lr', '--learning-rate', default=1e-3, type=float, metavar='LR',
↪ help='学习率，例如“1e-3” \n Learning rate, e.g., "1e-3": ', dest='lr')
parser.add_argument('--tau', default=2.0, type=float, help=
↪'LIF神经元的时间常数tau，例如“100.0” \n Membrane time constant, tau, for LIF␣
↪neurons, e.g., "100.0"')
parser.add_argument('-N', '--epoch', default=100, type=int, help=
↪'训练epoch，例如“100” \n Training epoch, e.g., "100"')
```

初始化数据加载器：

```
# 初始化数据加载器
train_dataset = torchvision.datasets.MNIST(
    root=dataset_dir,
    train=True,
    transform=torchvision.transforms.ToTensor(),
    download=True
)
test_dataset = torchvision.datasets.MNIST(
    root=dataset_dir,
    train=False,
    transform=torchvision.transforms.ToTensor(),
    download=True
)

train_data_loader = data.DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True,
    drop_last=True
)
test_data_loader = data.DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    shuffle=False,
    drop_last=False
)
```

定义并初始化网络：

```
# 定义并初始化网络
net = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 10, bias=False),
    neuron.LIFNode(tau=tau)
)
net = net.to(device)
```

初始化优化器、编码器（我们使用泊松编码器，将 MNIST 图像编码成脉冲序列）：

```
# 使用Adam优化器
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
# 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：

```
print("Epoch {}:".format(epoch))
print("Training...")
train_correct_sum = 0
train_sum = 0
net.train()
for img, label in tqdm(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())
```

(续下页)

(接上页)

```
    # 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
    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)

    # 正确率的计算方法如下。认为输出层中脉冲发放频率最大的神经元的下标i是分类结果
    train_correct_sum += (out_spikes_counter_frequency.max(1)[1] == label.to(device)).
↪float().sum().item()
    train_sum += label.numel()

    train_batch_accuracy = (out_spikes_counter_frequency.max(1)[1] == label.
↪to(device)).float().mean().item()
    writer.add_scalar('train_batch_accuracy', train_batch_accuracy, train_times)
    train_accs.append(train_batch_accuracy)

    train_times += 1
train_accuracy = train_correct_sum / train_sum
```

测试的代码与训练代码相比更为简单：

```
print("Testing...")
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 tqdm(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()
```

(续下页)

(接上页)

```
        functional.reset_net(net)
    test_accuracy = correct_sum / test_sum
    writer.add_scalar('test_accuracy', test_accuracy, epoch)
    test_accs.append(test_accuracy)
    max_test_accuracy = max(max_test_accuracy, test_accuracy)
print("Epoch {}: train_acc={}, test_acc={}, max_test_acc={}, train_times={}".
↪format(epoch, train_accuracy, test_accuracy, max_test_accuracy, train_times))
print()
```

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。可设置的（超）参数如下：

```
$ python <PATH>/lif_fc_mnist.py --help
usage: lif_fc_mnist.py [-h] [--device DEVICE] [--dataset-dir DATASET_DIR] [--log-dir␣
↪LOG_DIR] [--model-output-dir MODEL_OUTPUT_DIR] [-b BATCH_SIZE] [-T T] [--lr LR] [--
↪tau TAU] [-N EPOCH]

spikingjelly LIF MNIST Training

optional arguments:
-h, --help            show this help message and exit
--device DEVICE       运行的设备，例如“cpu”或“cuda:0” Device, e.g., "cpu" or
↪"cuda:0"
--dataset-dir DATASET_DIR
                      保存MNIST数据集的位置，例如“./” Root directory for saving␣
↪MNIST dataset, e.g., "./"
--log-dir LOG_DIR     保存tensorboard日志文件的位置，例如“./” Root directory for␣
↪saving tensorboard logs, e.g., "./"
--model-output-dir MODEL_OUTPUT_DIR
                      模型保存路径，例如“./” Model directory for saving, e.g., "./
↪"
-b BATCH_SIZE, --batch-size BATCH_SIZE
                      Batch 大小，例如“64” Batch size, e.g., "64"
-T T, --timesteps T   仿真时长，例如“100” Simulating timesteps, e.g., "100"
--lr LR, --learning-rate LR
                      学习率，例如“1e-3” Learning rate, e.g., "1e-3":
--tau TAU             LIF神经元的时间常数tau，例如“100.0” Membrane time constant,␣
↪tau, for LIF neurons, e.g., "100.0"
-N EPOCH, --epoch EPOCH
                      训练epoch，例如“100” Training epoch, e.g., "100"
```

也可以直接在 Python 命令行运行它：

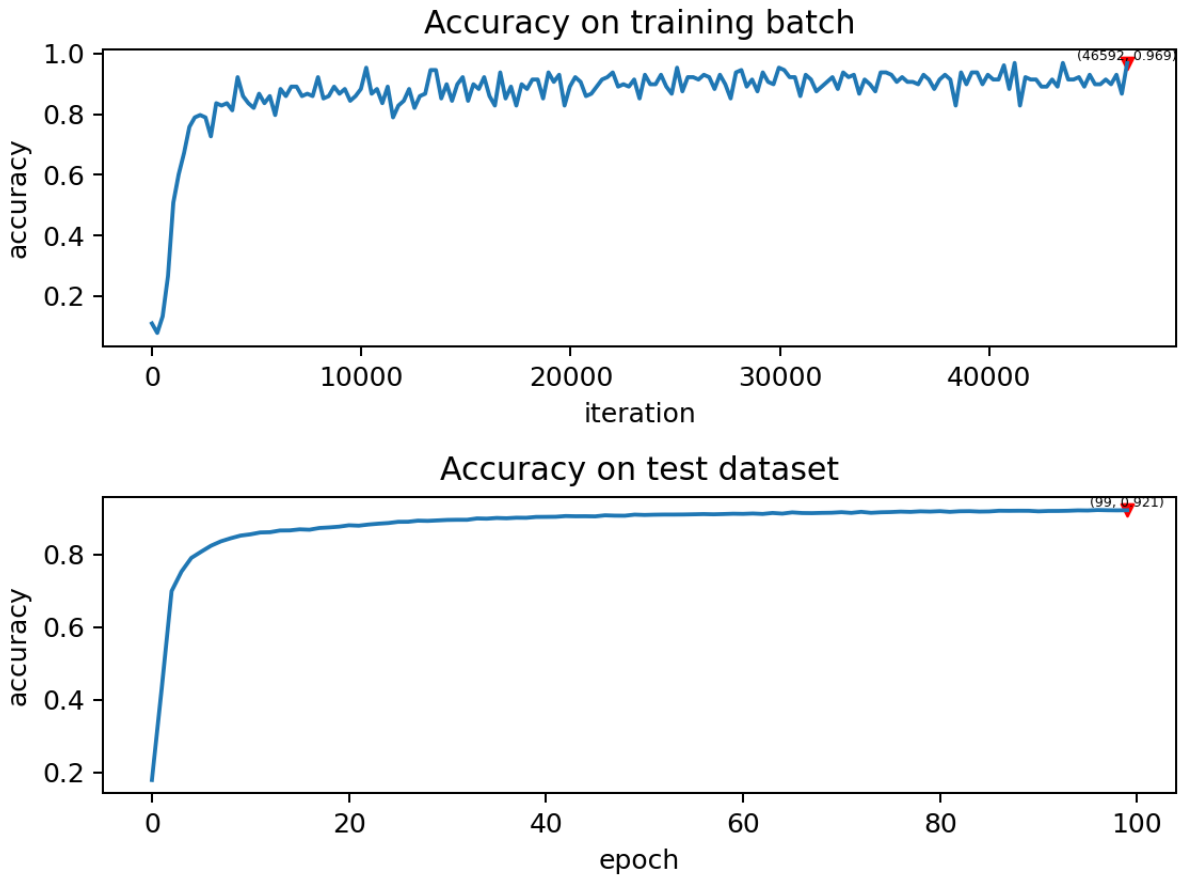
```
$ python
>>> import spikingjelly.clock_driven.examples.lif_fc_mnist as lif_fc_mnist
>>> lif_fc_mnist.main()
########## Configurations ##########
device=cuda:0
dataset_dir=./
log_dir=./
model_output_dir=./
batch_size=64
T=100
lr=0.001
tau=2.0
epoch=100
####################################
Epoch 0:
Training...
100
↪%|████████████████████████████████████████████████████████████████████████████████
↪937/937 [01:26<00:00, 10.89it/s]
Testing...
100
↪%|████████████████████████████████████████████████████████████████████████████████
↪157/157 [00:05<00:00, 28.79it/s]
Epoch 0: train_acc = 0.8641775613660619, test_acc=0.9071, max_test_acc=0.9071, train_
↪times=937
```

保存和读取模型：

```
# 保存模型
torch.save(net, model_output_dir + "/lif_snn_mnist.ckpt")
# 读取模型
# net = torch.load(model_output_dir + "/lif_snn_mnist.ckpt")
```

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())
```

(续下页)

(接上页)

```
    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)，[1] 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 示例：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 $X^i_{min} \leq X^i \leq X^i_{max}$，首先计算出 m 条高斯曲线 g^i_j 的均值和方差：

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

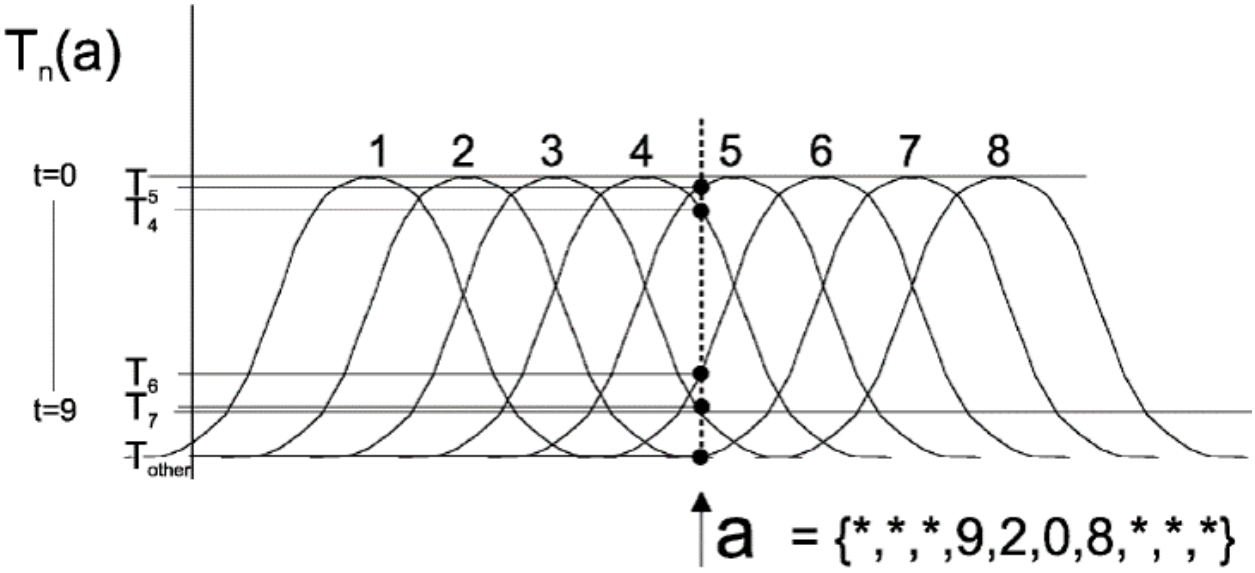
其中 β 通常取值为 1.5。可以看出，这 m 条高斯曲线的形状完全相同，只是对称轴所在的位置不同。

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[Page 14, 2] 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(28*28*m, 10, T)      # mnist 28*28=784
```

(续下页)

(接上页)

```
    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_batch_accuracy = (out_spikes_counter_frequency.max(1)[1] == label.to(device)).
↪float().mean().item()
```

我们使用的损失函数与Page 13, 1 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：

```
$ python
>>> import spikingjelly.event_driven.examples.tempotron_mnist as tempotron_mnist
>>> tempotron_mnist.main()
########## Configurations ##########
device=cuda:0
dataset_dir=./
log_dir=./
model_output_dir=./
batch_size=64
T=100
```

(续下页)

(接上页)

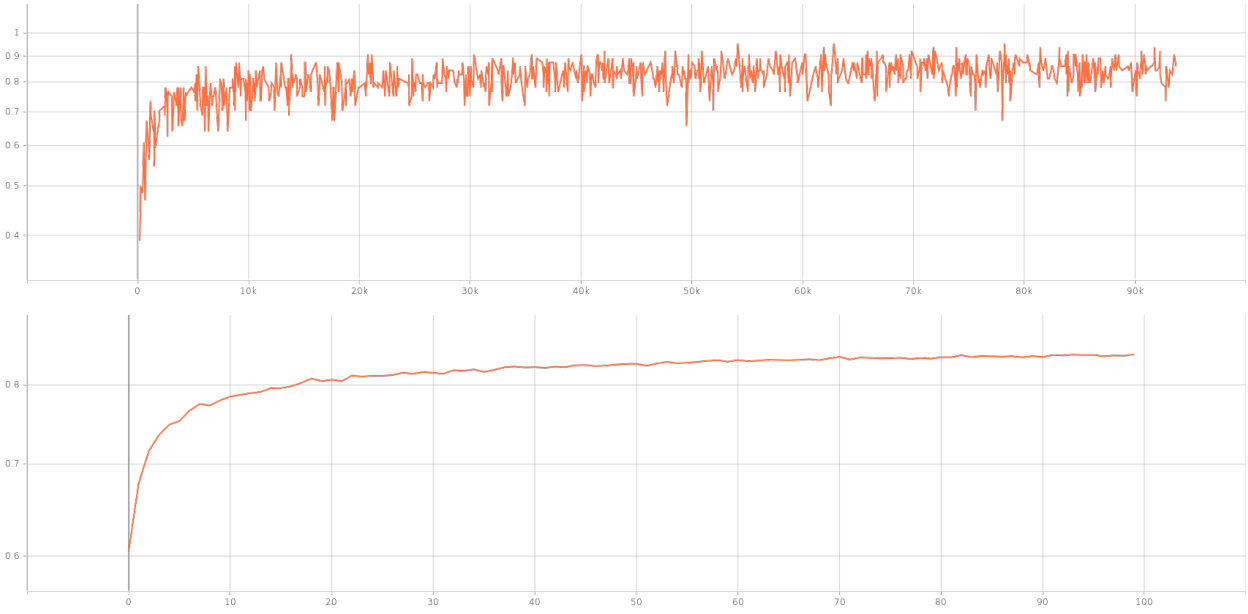
```
lr=0.001
epoch=100
m=16
####################################
Epoch 0:
Training...
100
↪%|████████████████████████████████████████████████████████████████████████████
↪937/937 [01:07<00:00, 13.91it/s]
Testing...
100
↪%|████████████████████████████████████████████████████████████████████████████
↪157/157 [00:10<00:00, 14.50it/s]
Epoch 0: train_acc = 0.49112860192102453, test_acc=0.6316, max_test_acc=0.6316, train_
↪times=937
```

保存和读取模型：

```
# 保存模型
torch.save(net, model_output_dir + "/tempotron_snn_mnist.ckpt")
# 读取模型
# net = torch.load(model_output_dir + "/tempotron_snn_mnist.ckpt")
```

查看训练结果

在 Tesla K80 上训练 100 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：



训练 100 个 Epoch，测试集的正确率为 84.19%，可以看出 Tempotron 实现了感知器的功能，具有一定的分类能力。

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, SNNs)。*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(tau=100.)
```

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
- **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

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.LIFNode* 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 `lif.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`，可以发现它与输入的数量是一致的：

```
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])
```

$V[t]$ 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）神经动态方程 $\frac{\mathrm{d}V(t)}{\mathrm{d}t} = f(V(t), X(t))$ 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*spikingjelly.clock_driven.neuron.LIFNode.neuronal_charge* 中找到如下所示的代码：

```
def neuronal_charge(self, x: torch.Tensor):
    if self.v_reset is None:
```

(续下页)

(接上页)

```
        self.v += (x - self.v) / self.tau

    else:
        if isinstance(self.v_reset, float) and self.v_reset == 0.:
            self.v += (x - self.v) / self.tau
        else:
            self.v += (x - (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*spikingjelly.clock_driven.neuron.BaseNode*，共享相同的放电、重置方程。可以在*spikingjelly.clock_driven.neuron.BaseNode.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：$V[t] = V_{reset}$
2. Soft 方式：释放脉冲后，膜电位减去阈值电压：$V[t] = V[t] - V_{threshold}$

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`spikingjelly.clock_driven.neuron.BaseNode.neuronal_fire.neuronal_reset`中可以找到膜电位重置的代码：

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

$$H[t] = f(V[t-1], X[t])$$
$$S[t] = g(H[t] - V_{threshold}) = \Theta(H[t] - V_{threshold})$$

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

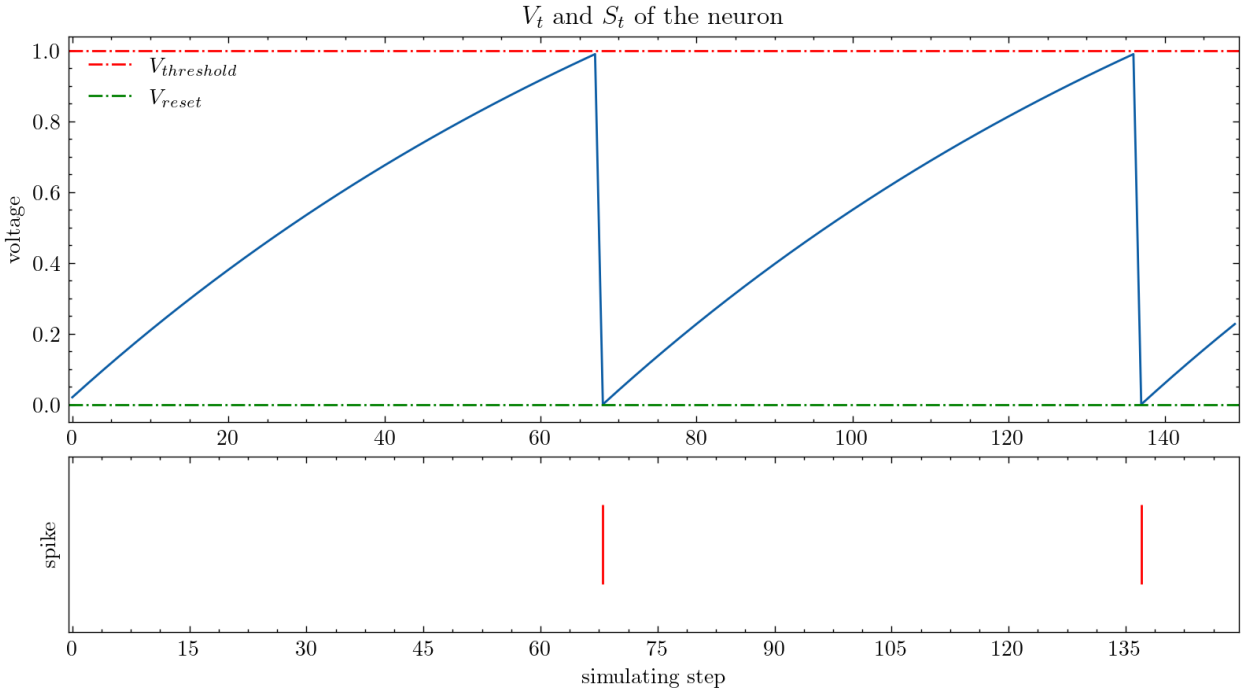
spikingjelly.clock_driven 使用时间驱动的方式，对 SNN 逐步进行仿真。

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。

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：

```
lif.reset()
x = torch.as_tensor([2.])
T = 150
s_list = []
v_list = []
for t in range(T):
    s_list.append(lif(x))
    v_list.append(lif.v)

visualizing.plot_one_neuron_v_s(np.asarray(v_list), np.asarray(s_list), v_
↪threshold=lif.v_threshold, v_reset=lif.v_reset,
                                dpi=200)
plt.show()
```

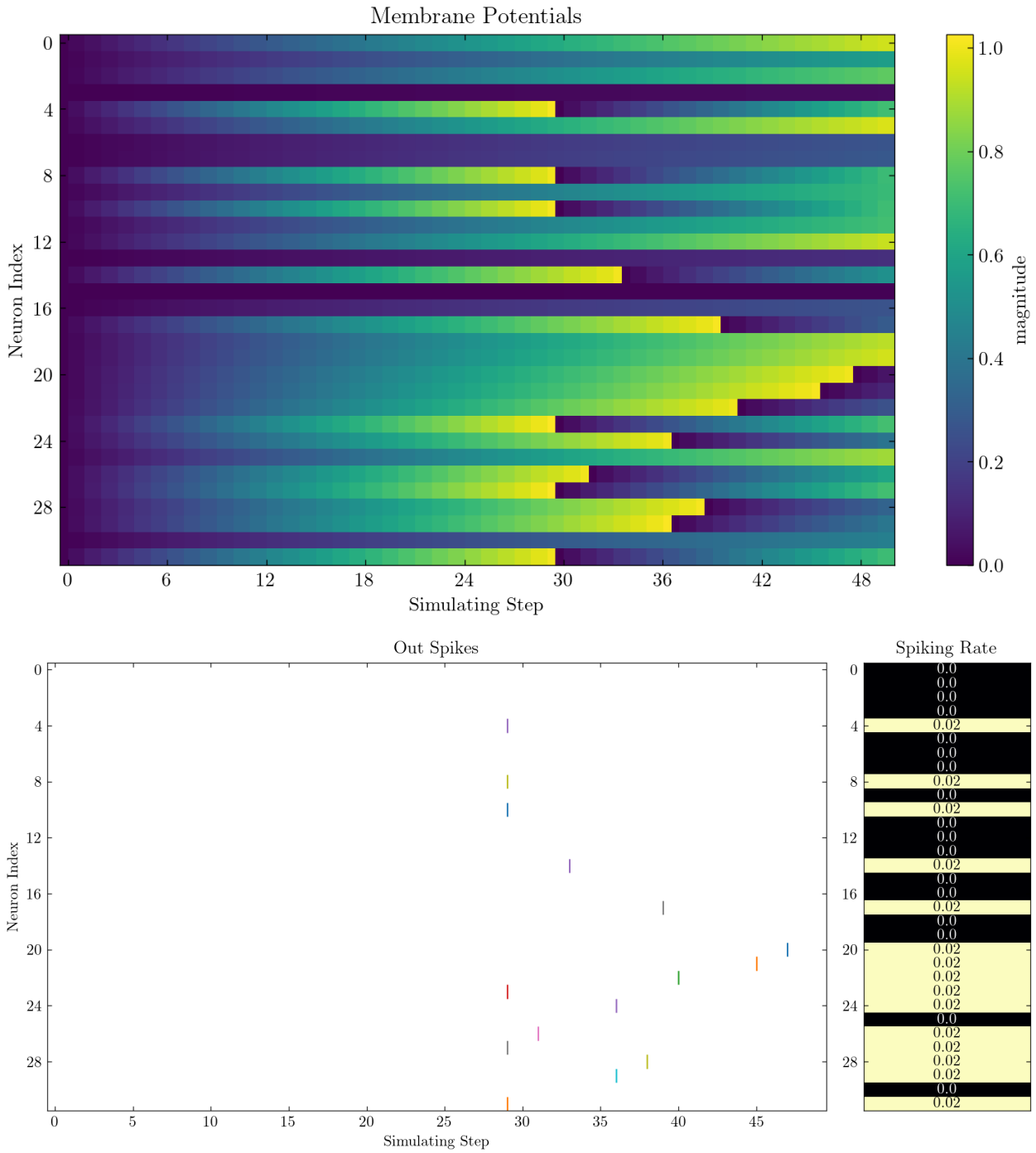
我们给与的输入 shape=[1]，因此这个 LIF 神经元层只有 1 个神经元。它的膜电位和输出脉冲随着时间变化情况如下：



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
s_list = []
v_list = []
for t in range(T):
    s_list.append(lif(x).unsqueeze(0))
    v_list.append(lif.v.unsqueeze(0))

s_list = torch.cat(s_list)
v_list = torch.cat(v_list)

visualizing.plot_2d_heatmap(array=np.asarray(v_list), 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(s_list), title='Membrane Potentials',␣
↪xlabel='Simulating Step',
                           ylabel='Neuron Index', dpi=200)
plt.show()
```

结果如下：


Membrane Potentials
Neuron Index
Simulating Step
magnitude
Out Spikes
Neuron Index
Simulating Step
Spiking Rate

1.4 时间驱动：编码器

本教程作者：Grasshlw, Yanqi-Chen, fangwei123456

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.encoding*中，存在 2 个基类编码器：

1. 无状态的编码器*spikingjelly.clock_driven.encoding.StatelessEncoder*
2. 有状态的编码器*spikingjelly.clock_driven.encoding.StatefulEncoder*

所定义的编码器都继承自这 2 个编码器基类之一。

无状态的编码器没有隐藏状态，输入数据`x[t]`可直接编码得到输出脉冲`spike[t]`；而有状态的编码器`encoder = StatefulEncoder(T)`，编码器会在首次调用`forward`时使用`encode`函数对`T`个时刻的输入序列`x`进行编码得到`spike`，在第`t`次调用`forward`时会输出`spike[t % T]`，可以从其前向传播的代码*spikingjelly.clock_driven.encoding.StatefulEncoder.forward*看到这种操作：

```
def forward(self, x: torch.Tensor):
    if self.spike is None:
        self.encode(x)

    t = self.t
    self.t += 1
    if self.t >= self.T:
        self.t = 0
    return self.spike[t]
```

与 SpikingJelly 中的其他有状态 module 一样，调用`reset()`函数可以将有状态编码器进行重新初始化。

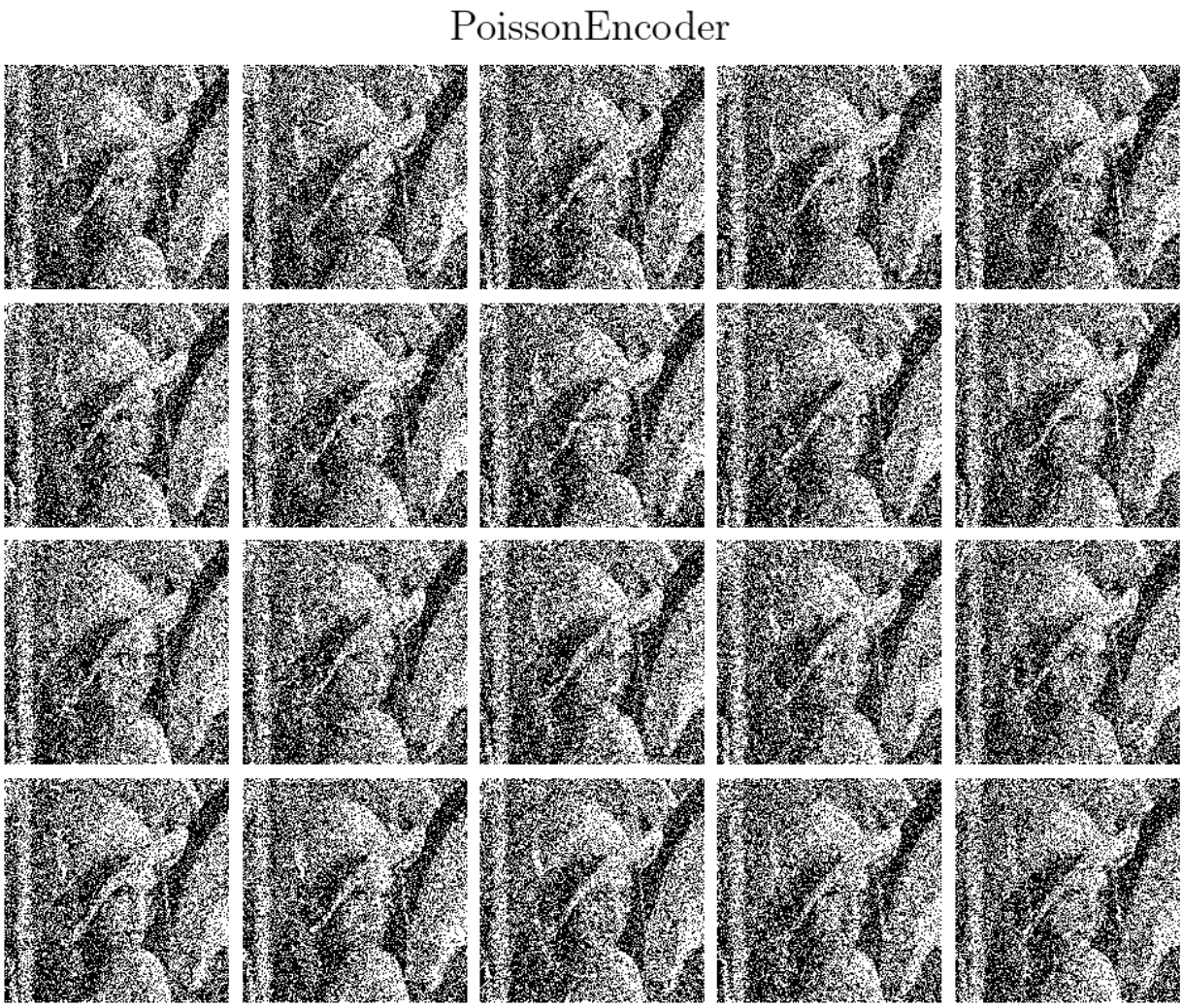
1.4.2 泊松编码器

泊松编码器*spikingjelly.clock_driven.encoding.PoissonEncoder*是无状态的编码器。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
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.imshow(x, cmap='gray')
plt.axis('off')

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

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):
```

(续下页)

(接上页)

```
    min_ = superposition_[i].min()
    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。

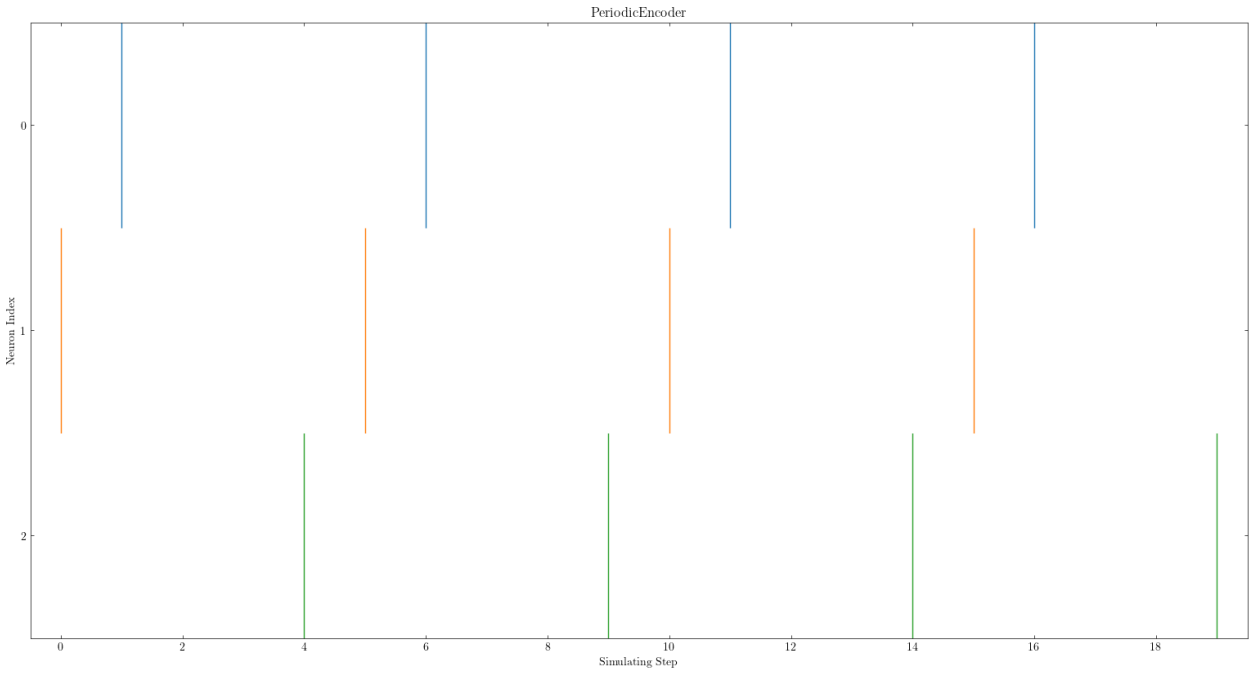
1.4.3 周期编码器

周期编码器*spikingjelly.clock_driven.encoding.PoissonEncoder*是周期性输出给定的脉冲序列的编码器。PeriodicEncoder 在初始化时可以设定好要输出的脉冲序列 spike ，也可以随时调用 spikingjelly.clock_driven.encoding.PoissonEncoder.encode 重新设定。

```
class PeriodicEncoder(BaseEncoder):
    def __init__(self, spike: torch.Tensor):
        super().__init__(spike.shape[0])
        self.encode(spike)
    def encode(self, spike: torch.Tensor):
        self.spike = spike
        self.T = spike.shape[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。

```
spike = torch.full((5, 3), 0)
spike[1, 0] = 1
spike[0, 1] = 1
spike[4, 2] = 1

pe = encoding.PeriodicEncoder(spike)

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

visualizing.plot_1d_spikes(out_spike.float().numpy(), 'PeriodicEncoder', 'Simulating␣
↪Step', 'Neuron Index',
                           plot_firing_rate=False)
plt.show()
```



1.4.4 延迟编码器

延迟编码器*spikingjelly.clock_driven.encoding.LatencyEncoder* 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_f 与刺激强度 $x \in [0, 1]$ 满足以下二式：当编码类型为线性时（`function_type='linear'`）

$$t_f(x) = (T - 1)(1 - x)$$

当编码类型为对数时（`function_type='log'`）

$$t_f(x) = (T-1) - ln(\alpha * x + 1)$$

其中，T 为最大脉冲发放时间，x 需归一化到 $[0, 1]$。

考虑第二个式子，α 需满足：

$$(T-1) - ln(\alpha * 1 + 1) = 0$$

这会导致该编码器很可能发生溢出，因为

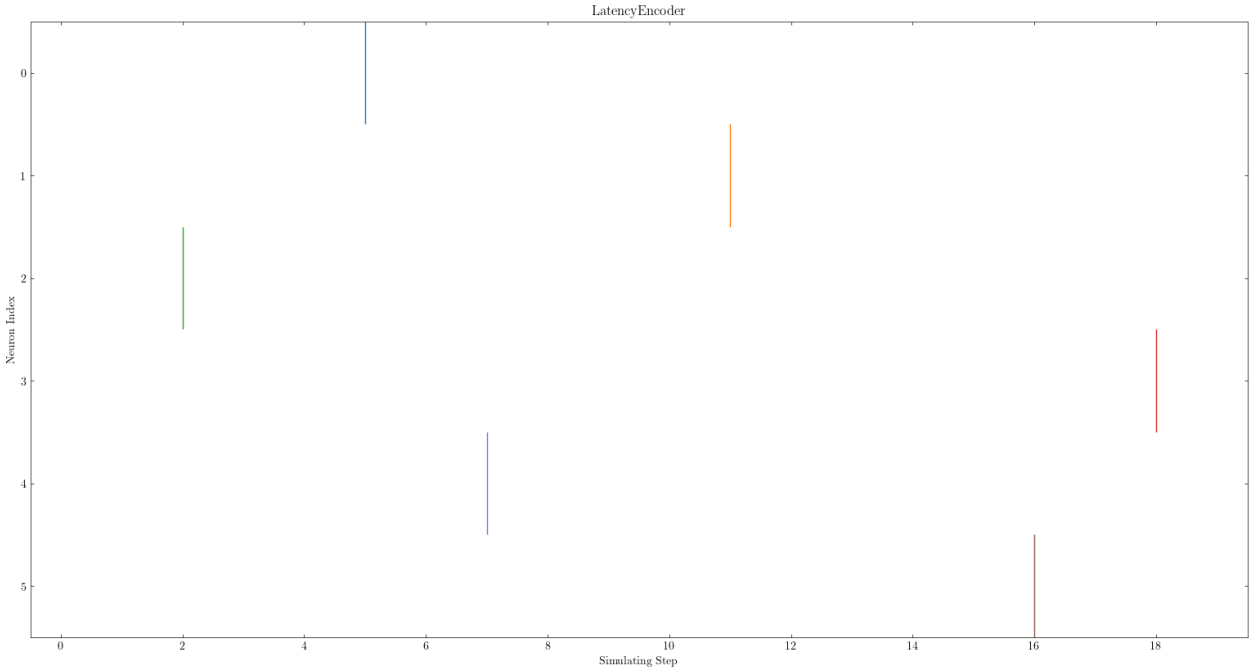
$$\alpha = e^{T-1} - 1$$

α 会随着 T 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.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
N = 6
x = torch.rand([N])
T = 20

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

# 输出延迟编码器的编码结果
out_spike = torch.zeros([T, N])
for t in range(T):
    out_spike[t] = le(x)

print(x)
visualizing.plot_1d_spikes(out_spike.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：



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

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：

```
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如学习率等以及若干其他配置

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=lr)
# 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：

```
net.train()
print("Epoch {}:".format(epoch))
print("Training...")
for img, label in tqdm(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
    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)

    # 正确率的计算方法如下。认为输出层中脉冲发放频率最大的神经元的下标i是分类结果
    train_accuracy = (out_spikes_counter_frequency.max(1)[1] == label.to(device)).
↪float().mean().item()

    writer.add_scalar('train_accuracy', train_accuracy, train_times)
```

(续下页)

(接上页)

```
    train_accs.append(train_accuracy)

    train_times += 1
```

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。可以直接在命令行运行它：

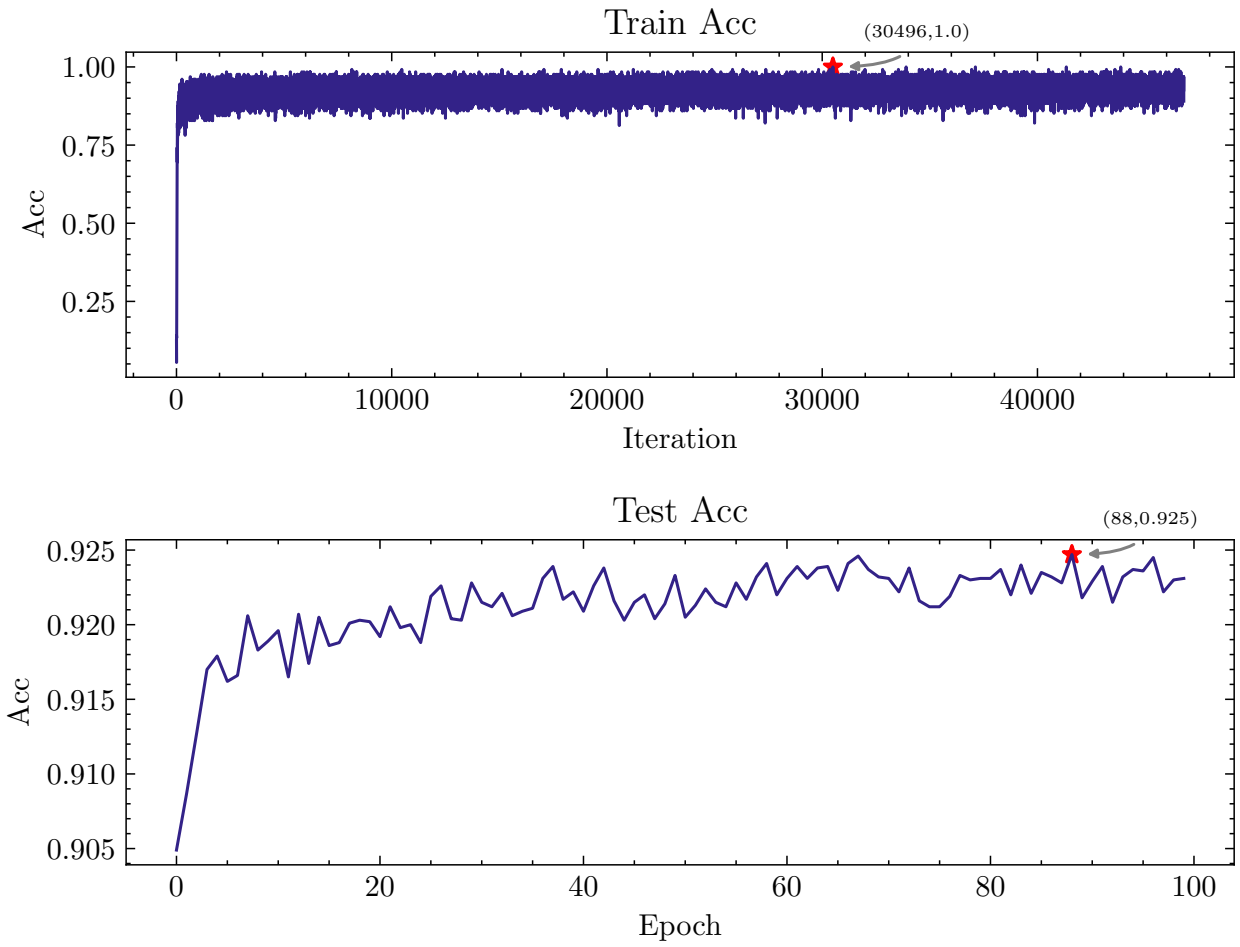
```
$ python <PATH>/lif_fc_mnist.py --help
usage: lif_fc_mnist.py [-h] [--device DEVICE] [--dataset-dir DATASET_DIR] [--log-dir␣
↪LOG_DIR] [--model-output-dir MODEL_OUTPUT_DIR] [-b BATCH_SIZE] [-T T] [--lr LR] [--
↪tau TAU] [-N EPOCH]

spikingjelly LIF MNIST Training

optional arguments:
-h, --help            show this help message and exit
--device DEVICE       运行的设备，例如“cpu”或“cuda:0” Device, e.g., "cpu" or
↪"cuda:0"
--dataset-dir DATASET_DIR
                      保存MNIST数据集的位置，例如“./” Root directory for saving␣
↪MNIST dataset, e.g., "./"
--log-dir LOG_DIR     保存tensorboard日志文件的位置，例如“./” Root directory for␣
↪saving tensorboard logs, e.g., "./"
--model-output-dir MODEL_OUTPUT_DIR
                      模型保存路径，例如“./” Model directory for saving, e.g., "./
↪"
-b BATCH_SIZE, --batch-size BATCH_SIZE
                      Batch 大小，例如“64” Batch size, e.g., "64"
-T T, --timesteps T   仿真时长，例如“100” Simulating timesteps, e.g., "100"
--lr LR, --learning-rate LR
                      学习率，例如“1e-3” Learning rate, e.g., "1e-3":
--tau TAU             LIF神经元的时间常数tau，例如“100.0” Membrane time constant,␣
↪tau, for LIF neurons, e.g., "100.0"
-N EPOCH, --epoch EPOCH
                      训练epoch，例如“100” Training epoch, e.g., "100"
```

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。

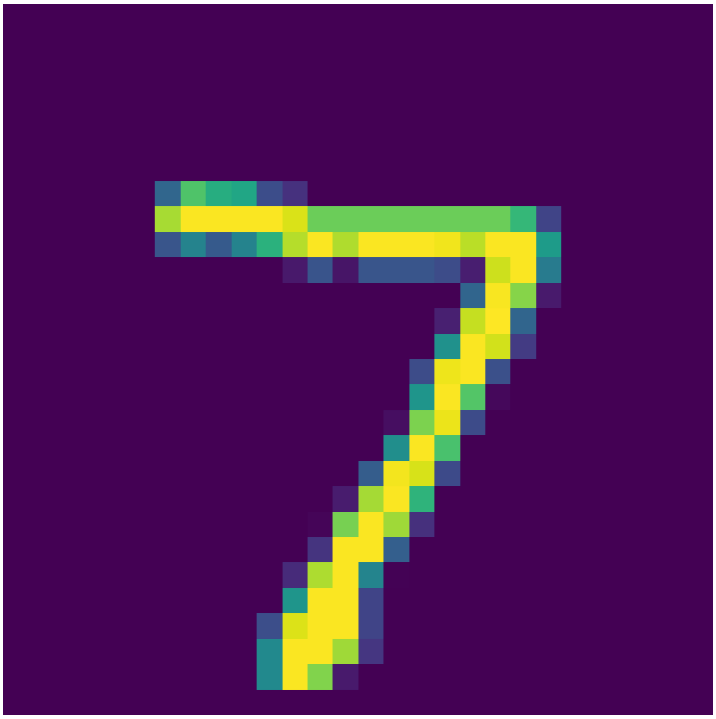
另外由于我们使用了泊松编码器，因此需要较大的 T。

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



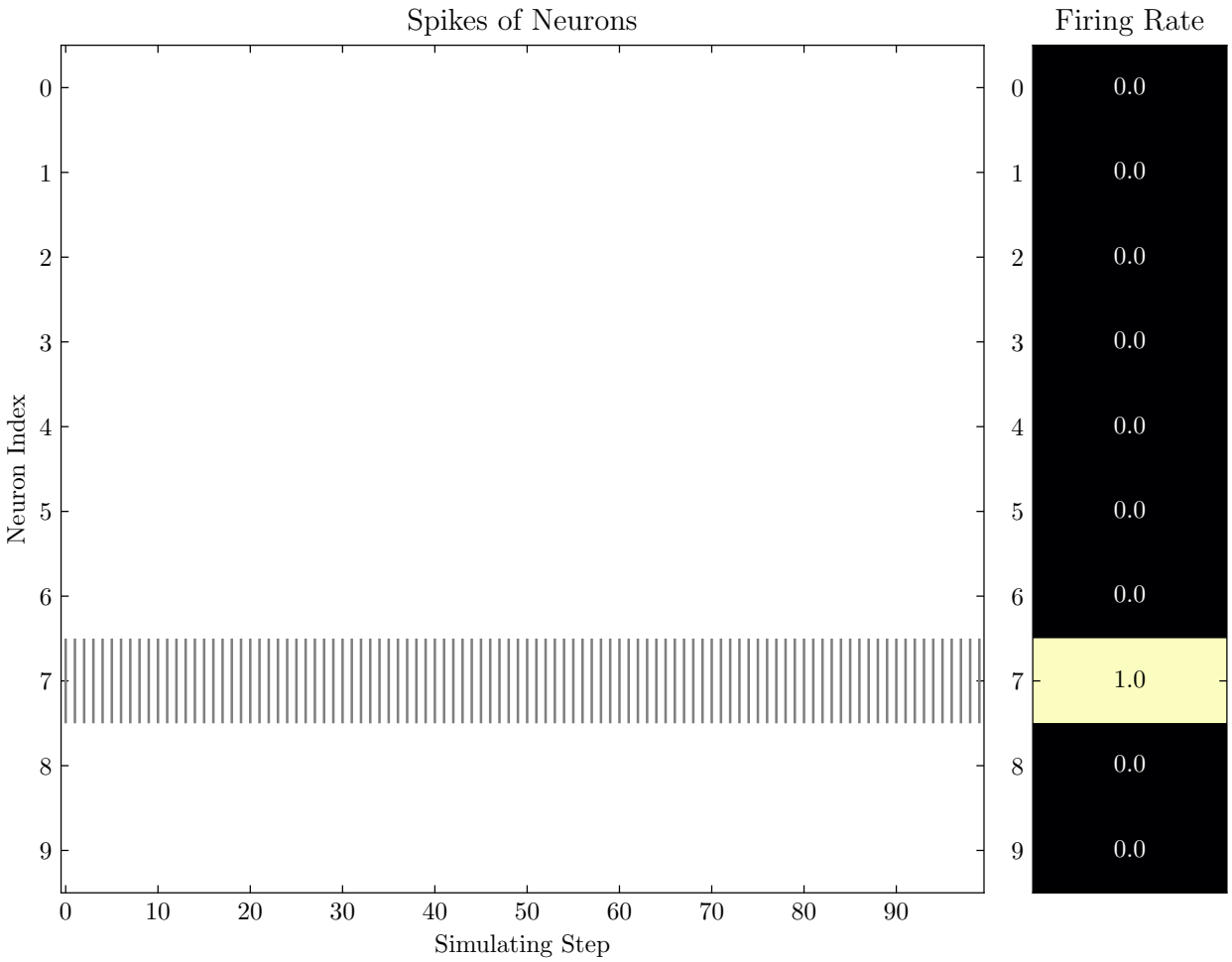
选取测试集中第一张图片：

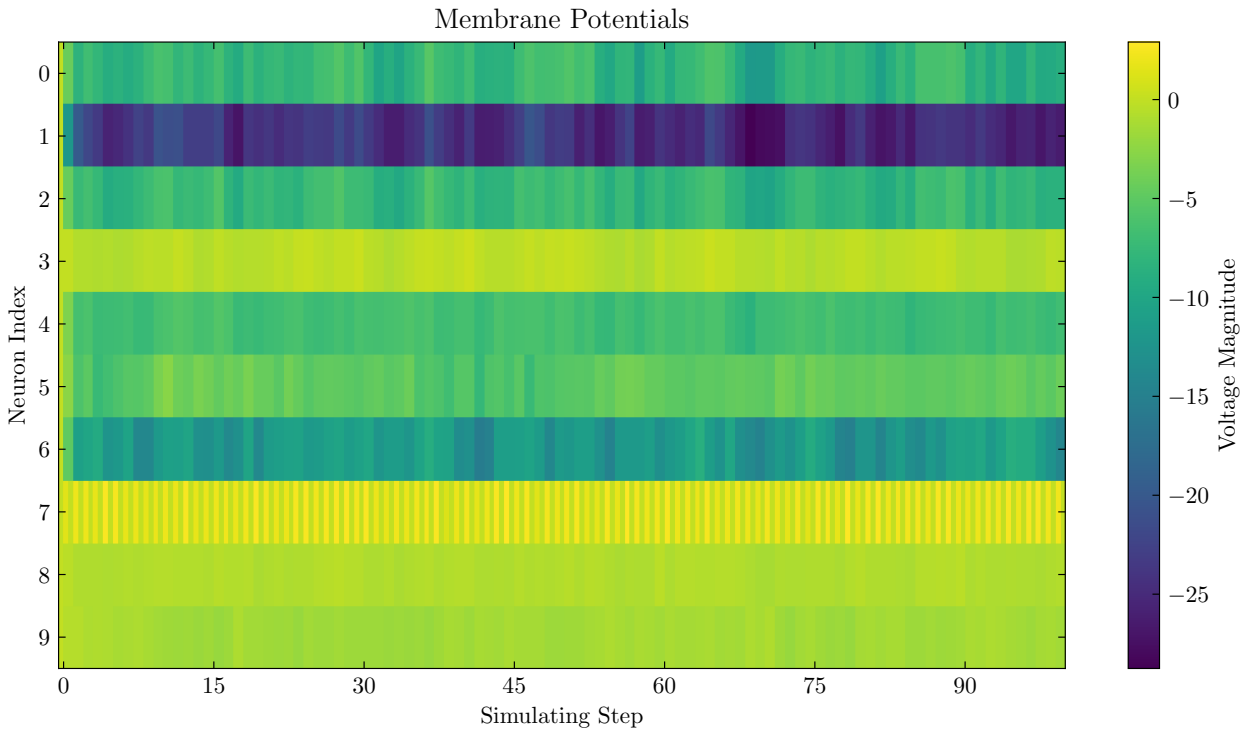


用训好的模型进行分类，得到分类结果

```
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
0.0
0.0
0.0
0.0
0.0
0.0
1.0
0.0
0.0
0
10
20
30
40
50
60
70
80
90



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 os
import time
```

(续下页)

(接上页)

```
import argparse
import numpy as np
from torch.cuda import amp
_seed_ = 2020
torch.manual_seed(_seed_)  # use torch.manual_seed() to seed the RNG for all devices␣
↪(both CPU and CUDA)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
np.random.seed(_seed_)

class PythonNet(nn.Module):
    def __init__(self, T):
        super().__init__()
        self.T = T
```

接下来，我们在 PythonNet 的成员变量中添加卷积层和全连接层。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(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(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。具体而言，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(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 2 层全连接网络，输出分类的结果。Fashion-MNIST 共有 10 类，因此输出层是 10 个神经元。

```
self.fc = nn.Sequential(
    nn.Flatten(),
    nn.Linear(128 * 7 * 7, 128 * 4 * 4, bias=False),
    neuron.IFNode(surrogate_function=surrogate.ATan()),
    nn.Linear(128 * 4 * 4, 10, bias=False),
    neuron.IFNode(surrogate_function=surrogate.ATan()),
)
```

接下来，定义前向传播：

```
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
```

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。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(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(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。我们可以提取出这 2 层，不参与时间上的循环。完整的代码如下：

```
class PythonNet(nn.Module):
    def __init__(self, T):
        super().__init__()
        self.T = T
```

(续下页)

(接上页)

```
        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(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(surrogate_function=surrogate.ATan()),
            nn.MaxPool2d(2, 2)  # 7 * 7

        )
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128 * 7 * 7, 128 * 4 * 4, bias=False),
            neuron.IFNode(surrogate_function=surrogate.ATan()),
            nn.Linear(128 * 4 * 4, 10, bias=False),
            neuron.IFNode(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*，训练命令如下：

```
Classify Fashion-MNIST

optional arguments:
  -h, --help            show this help message and exit
  -T T                  simulating time-steps
  -device DEVICE        device
  -b B                  batch size
  -epochs N             number of total epochs to run
  -j N                  number of data loading workers (default: 4)
  -data_dir DATA_DIR    root dir of Fashion-MNIST dataset
  -out_dir OUT_DIR      root dir for saving logs and checkpoint
  -resume RESUME        resume from the checkpoint path
  -amp                  automatic mixed precision training
  -cupy                 use cupy neuron and multi-step forward mode
  -opt OPT              use which optimizer. SDG or Adam
  -lr LR                learning rate
  -momentum MOMENTUM    momentum for SGD
  -lr_scheduler LR_SCHEDULER
                        use which schedule. StepLR or CosALR
  -step_size STEP_SIZE  step_size for StepLR
  -gamma GAMMA          gamma for StepLR
  -T_max T_MAX          T_max for CosineAnnealingLR
```

其中 `-cupy` 是使用 cupy 后端和多步神经元，关于它的更多信息参见传播模式 和使用 *CUDA* 增强的神经元与逐层传播进行加速。

检查点会被保存在 `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。

```
(pytorch-env) root@e8b6e4800dae4011eb0918702bd7ddedd51c-fangw1598-0:/# python -m␣
↪spikingjelly.clock_driven.examples.conv_fashion_mnist -opt SGD -data_dir /userhome/
↪datasets/FashionMNIST/ -amp

Namespace(T=4, T_max=64, amp=True, b=128, cupy=False, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
PythonNet(
  (static_conv): Sequential(
    (0): Conv2d(1, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
```

(续下页)

(接上页)

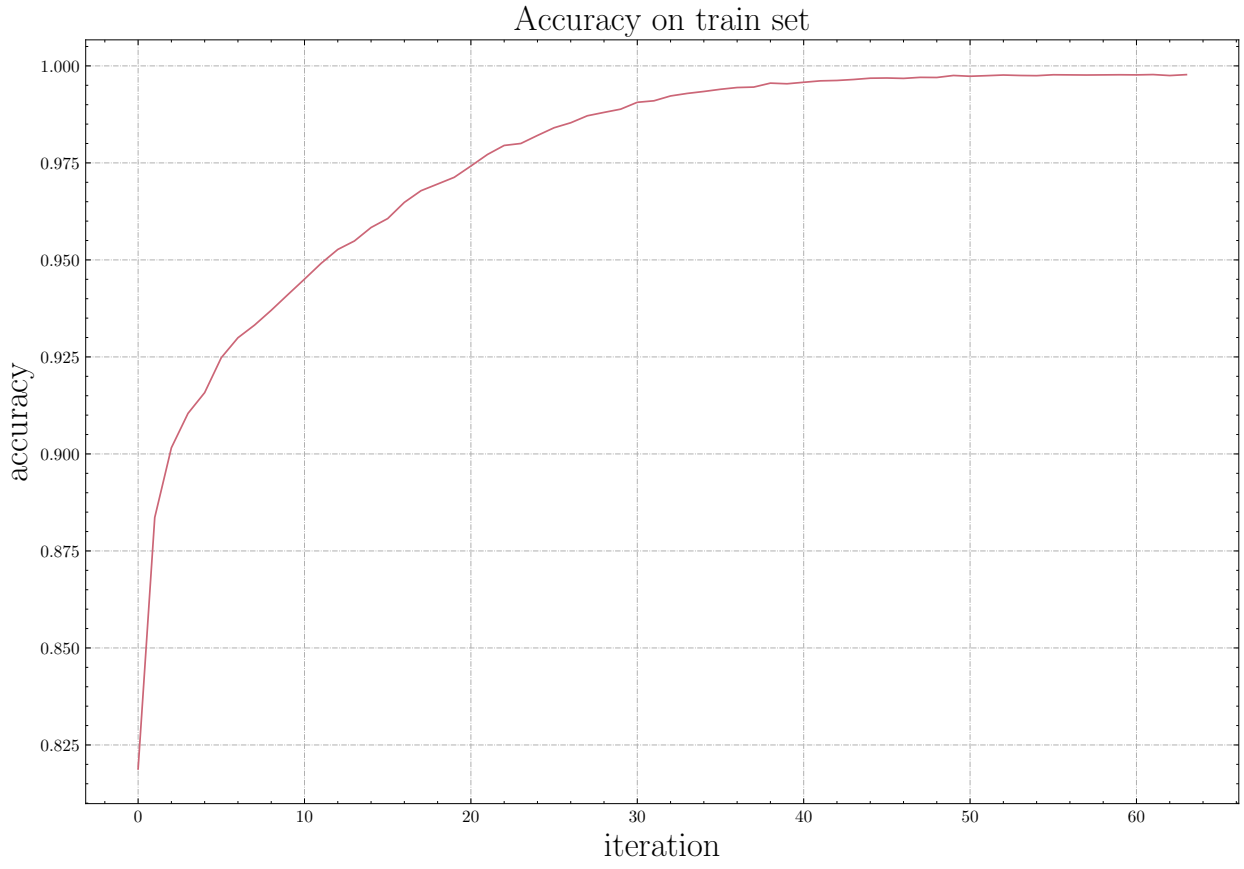
```
  )
  (conv): Sequential(
    (0): IFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
    (4): IFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=6272, out_features=2048, bias=False)
    (2): IFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (3): Linear(in_features=2048, out_features=10, bias=False)
    (4): IFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
  )
)
Mkdir ./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp.
Namespace(T=4, T_max=64, amp=True, b=128, cupy=False, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp
epoch=0, train_loss=0.028124165828697957, train_acc=0.8188267895299145, test_loss=0.
↪023525000348687174, test_acc=0.8633, max_test_acc=0.8633, total_time=16.
↪86261749267578
Namespace(T=4, T_max=64, amp=True, b=128, cupy=False, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp
```

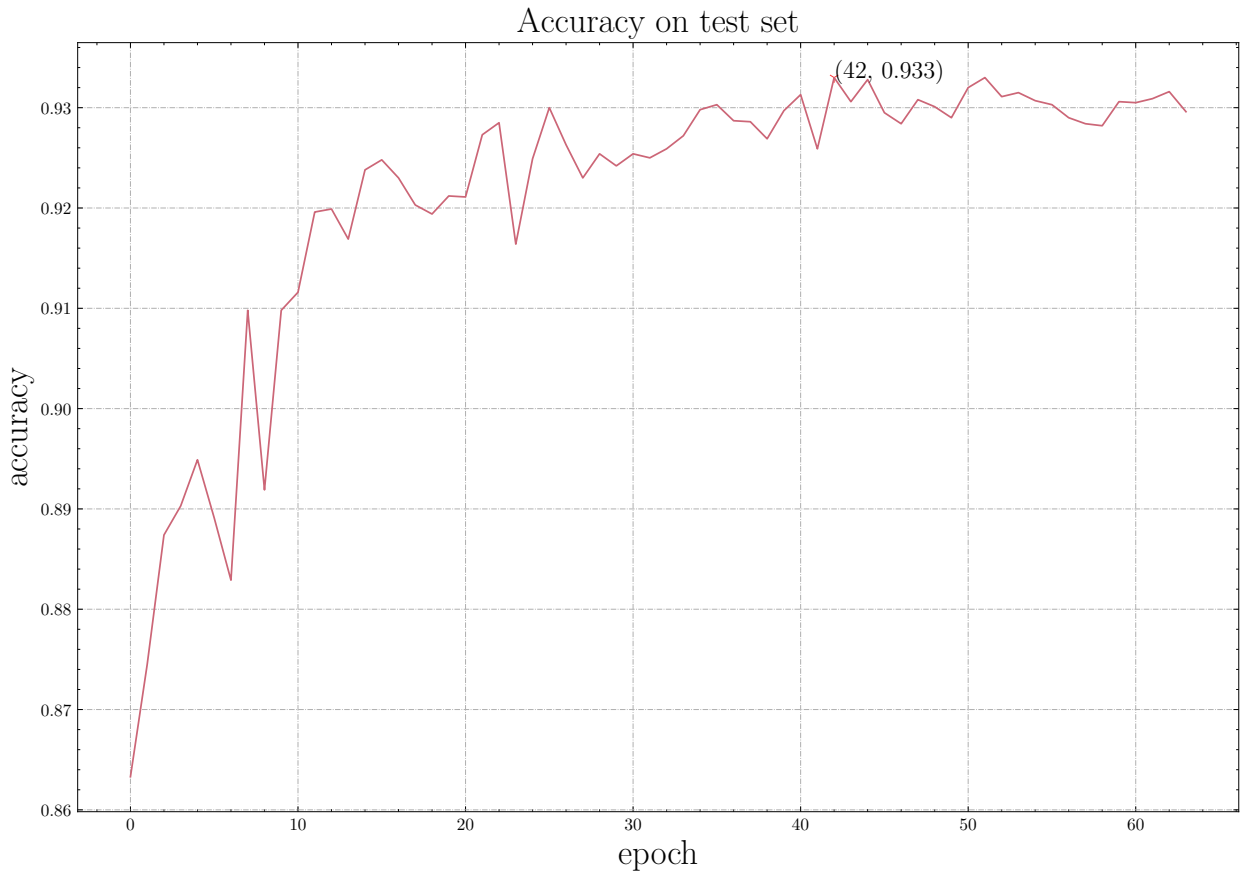
(续下页)

(接上页)

```
epoch=1, train_loss=0.018544567498163536, train_acc=0.883613782051282, test_loss=0.
↪02161250041425228, test_acc=0.8745, max_test_acc=0.8745, total_time=16.
↪618073225021362
Namespace(T=4, T_max=64, amp=True, b=128, cupy=False, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)

...

./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp
epoch=62, train_loss=0.0010829827882937538, train_acc=0.997512686965812, test_loss=0.
↪011441250185668468, test_acc=0.9316, max_test_acc=0.933, total_time=15.
↪976636171340942
Namespace(T=4, T_max=64, amp=True, b=128, cupy=False, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp
epoch=63, train_loss=0.0010746361010835525, train_acc=0.9977463942307693, test_loss=0.
↪01154562517106533, test_acc=0.9296, max_test_acc=0.933, total_time=15.83976149559021
```

运行 64 轮训练后，训练集和测试集上的正确率如下：





在训练 64 个 epoch 后，最高测试集正确率可以达到 93.3%，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, 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(surrogate_function=surrogate.ATan()),
        ...
```

(续下页)

(接上页)

```
        )
```

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 PythonNet
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/T_4_b_128_SGD_lr_0.1_CosALR_64_amp/checkpoint_max.pth', 'cpu
↪')['net']
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]。

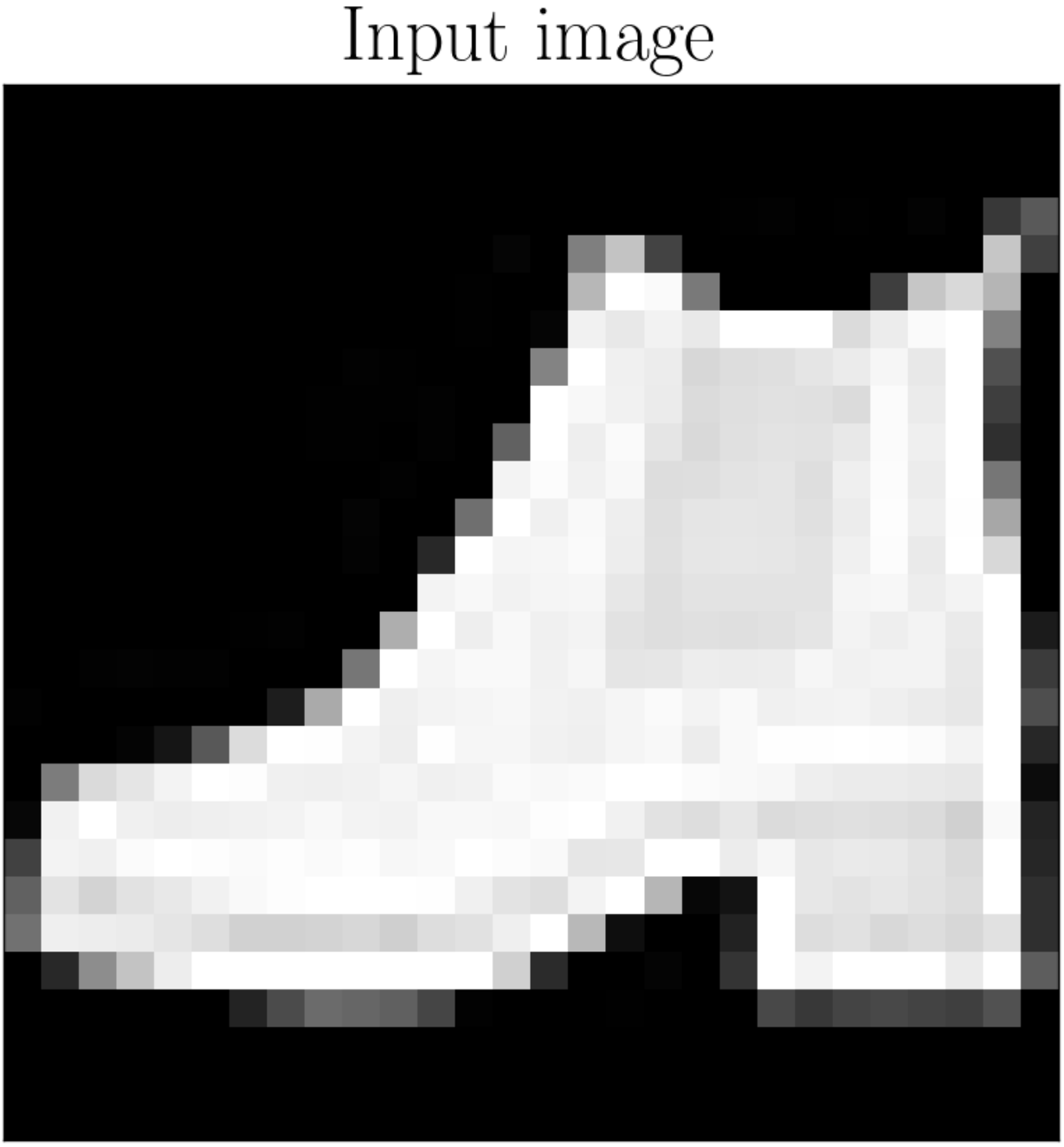
```
with torch.no_grad():
    # 每遍历一次全部数据集，就在测试集上测试一次
    for img, label in test_data_loader:
        fig = plt.figure(dpi=200)
        plt.imshow(img.squeeze().numpy(), cmap='gray')
        # 注意输入到网络的图片尺寸是 ``[1, 1, 28, 28]``，第0个维度是␣
↪``batch``，第1个维度是 ``channel``
        # 因此在调用 ``imshow`` 时，先使用 ``squeeze()``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

(续下页)

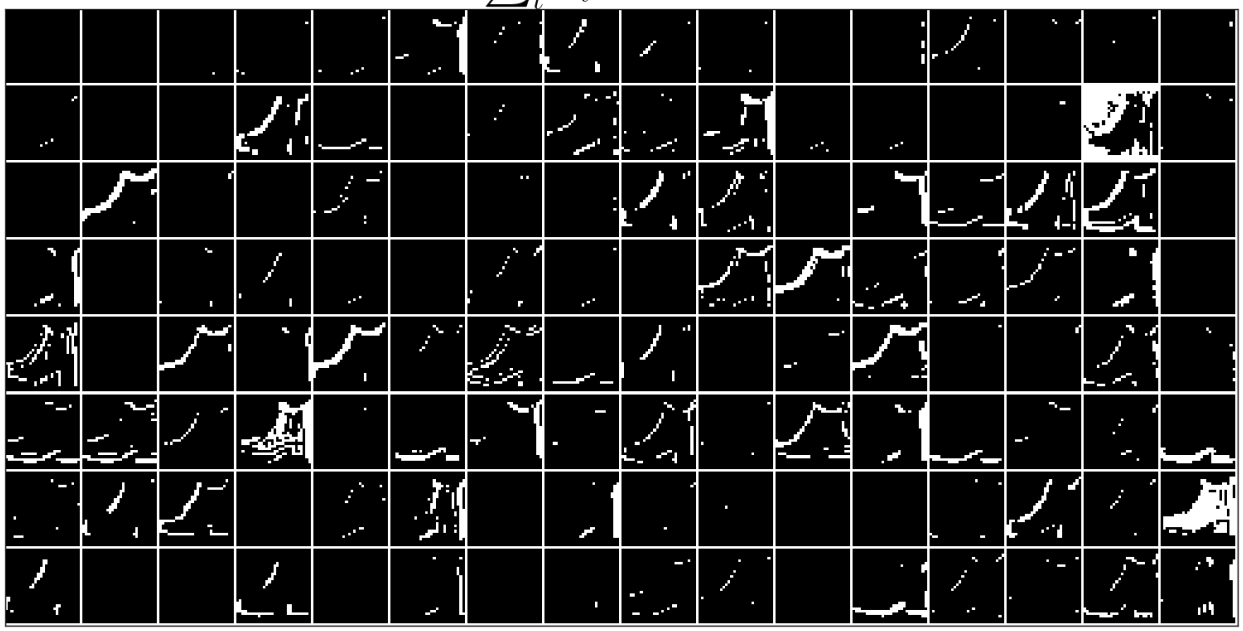
(接上页)

```
        plt.title('Input image', fontsize=20)
        plt.xticks([])
        plt.yticks([])
        plt.show()
        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()
```

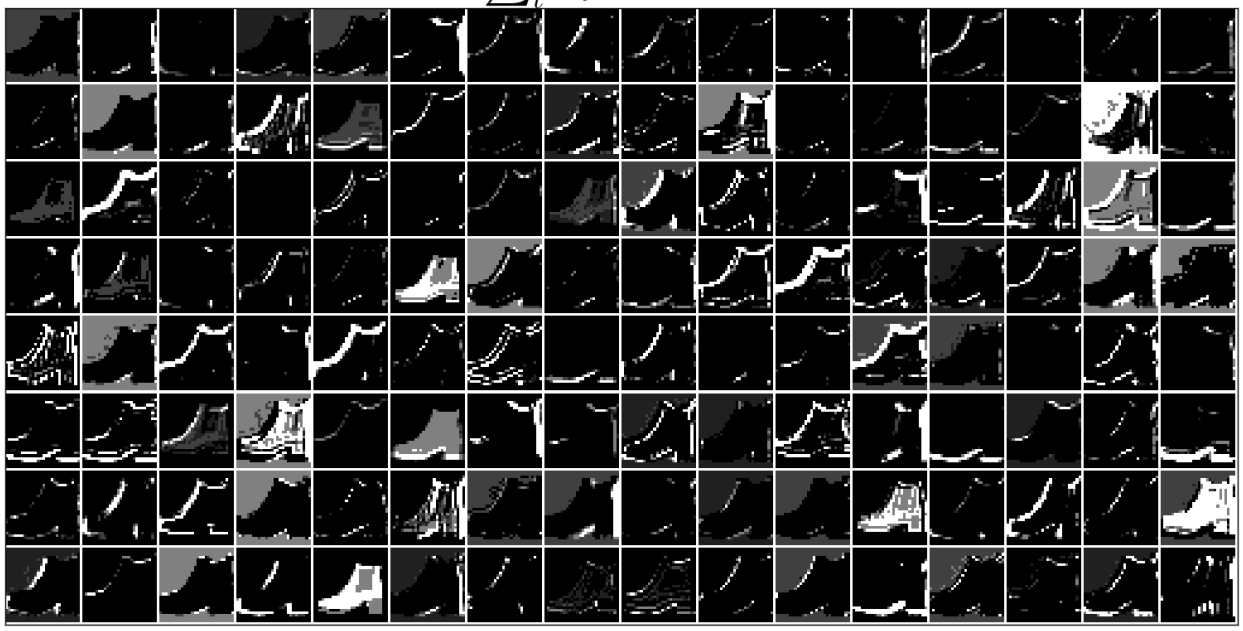
下面展示 2 个输入图片，以及在最开始 t=0 和最后 t=7 时刻的编码器输出的累计脉冲 $\sum_t S_t$：

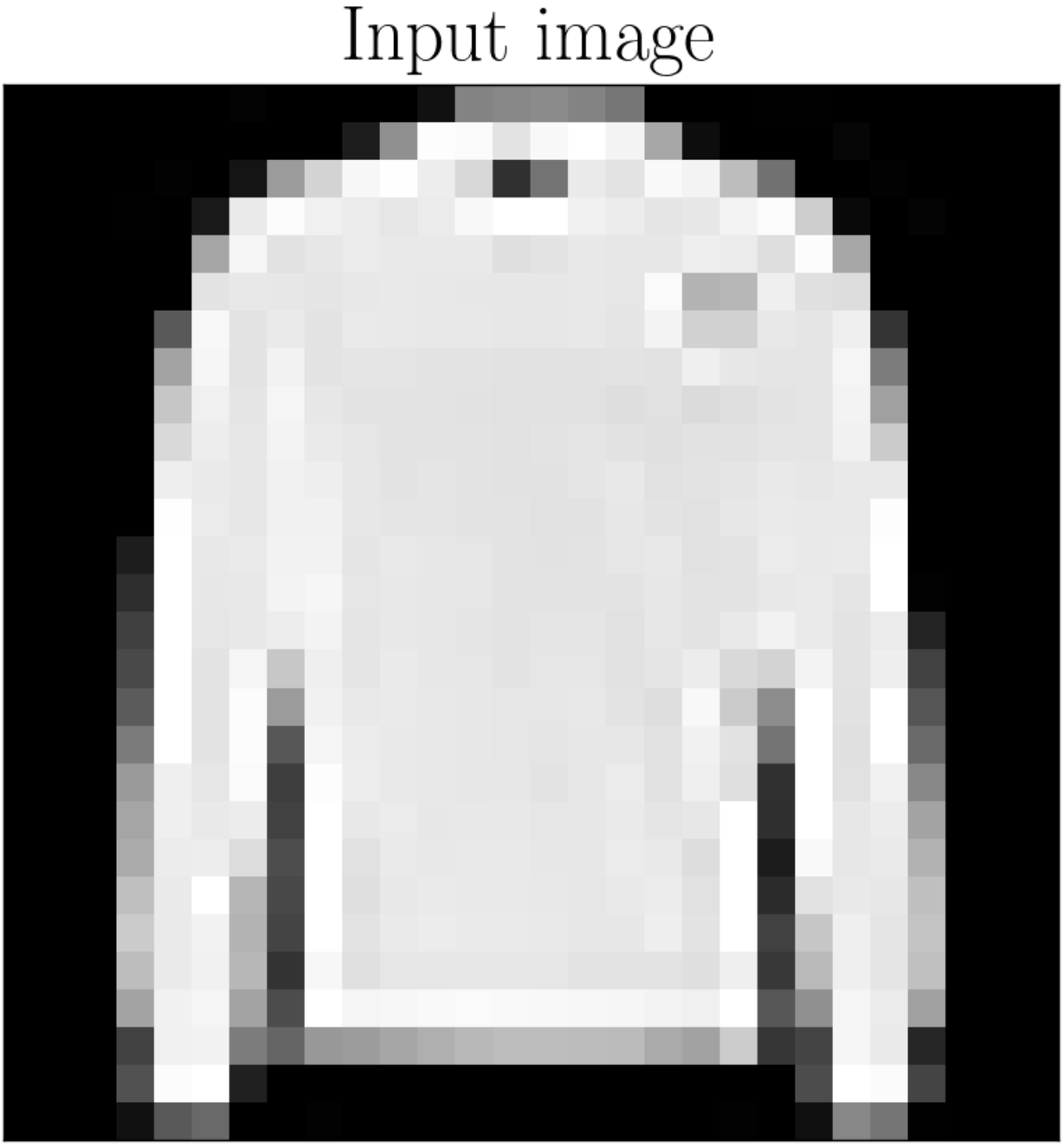

Input image

$\sum_t S_t$ at $t = 0$

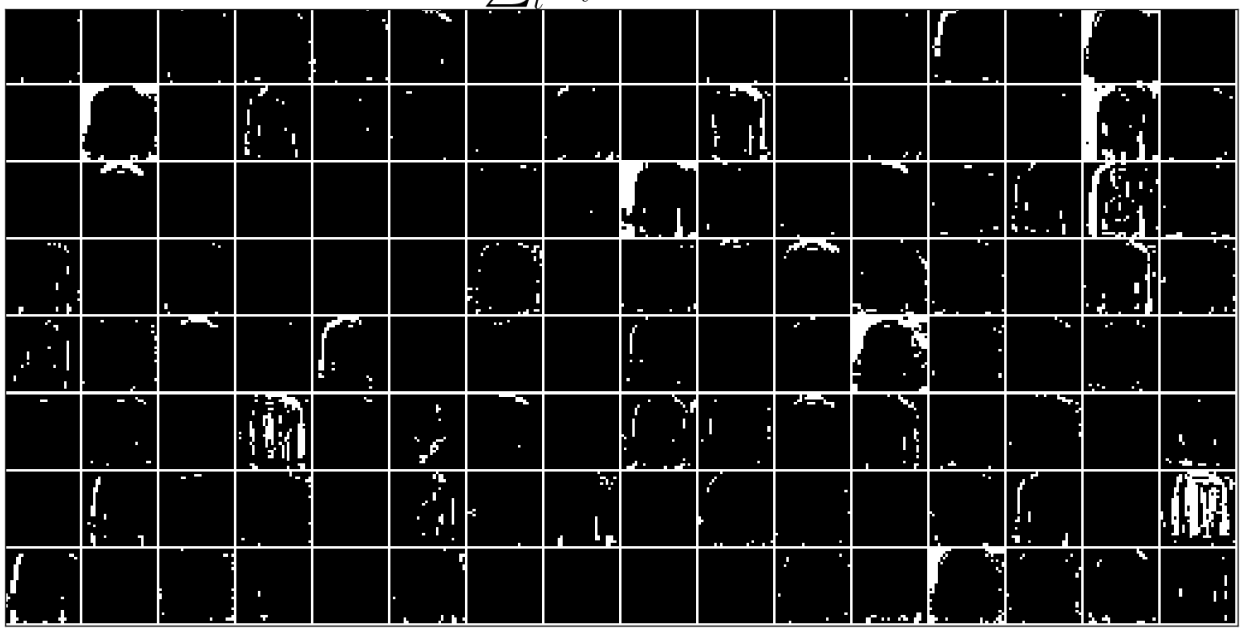


$\sum_t S_t$ at $t = 7$

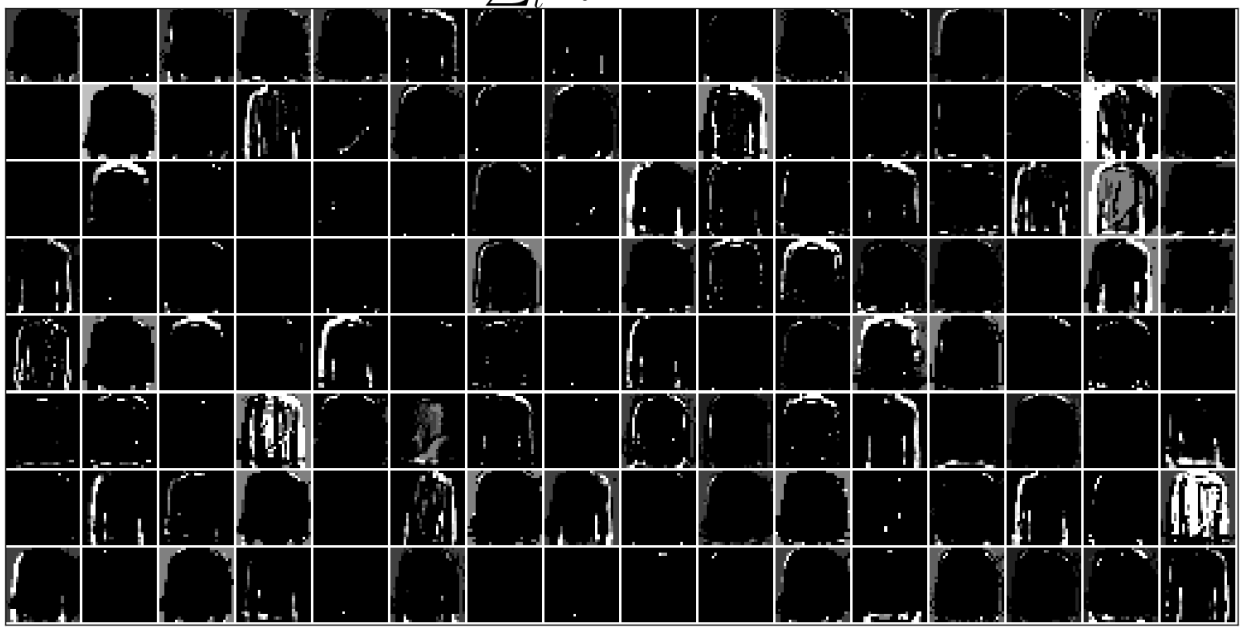



Input image

$\sum_t S_t$ 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，表明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。

较早的实现方案中有两套实现：基于 ONNX 和基于 PyTorch。由于 ONNX 不稳定，本版本为 PyTorch 增强版，原生支持复杂拓扑（例如 ResNet）。一起来看看吧！

1.7.1 ANN 转换 SNN 的理论基础

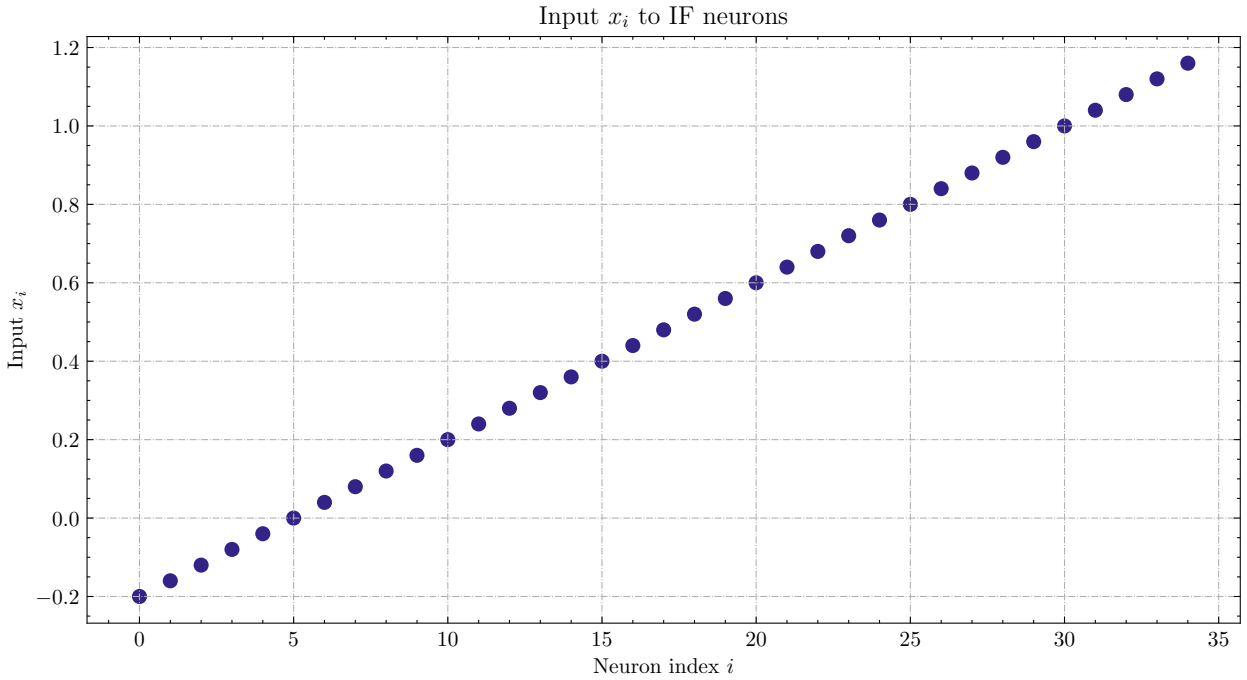
SNN 相比于 ANN，产生的脉冲是离散的，这有利于高效的通信。在 ANN 大行其道的今天，SNN 的直接训练需要较多资源。自然我们会想到使用现在非常成熟的 ANN 转换到 SNN，希望 SNN 也能有类似的表现。这就牵扯到如何搭建起 ANN 和 SNN 桥梁的问题。现在 SNN 主流的方式是采用频率编码，因此对于输出层，我们会用神经元输出脉冲数来判断类别。发放率和 ANN 有没有关系呢?

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

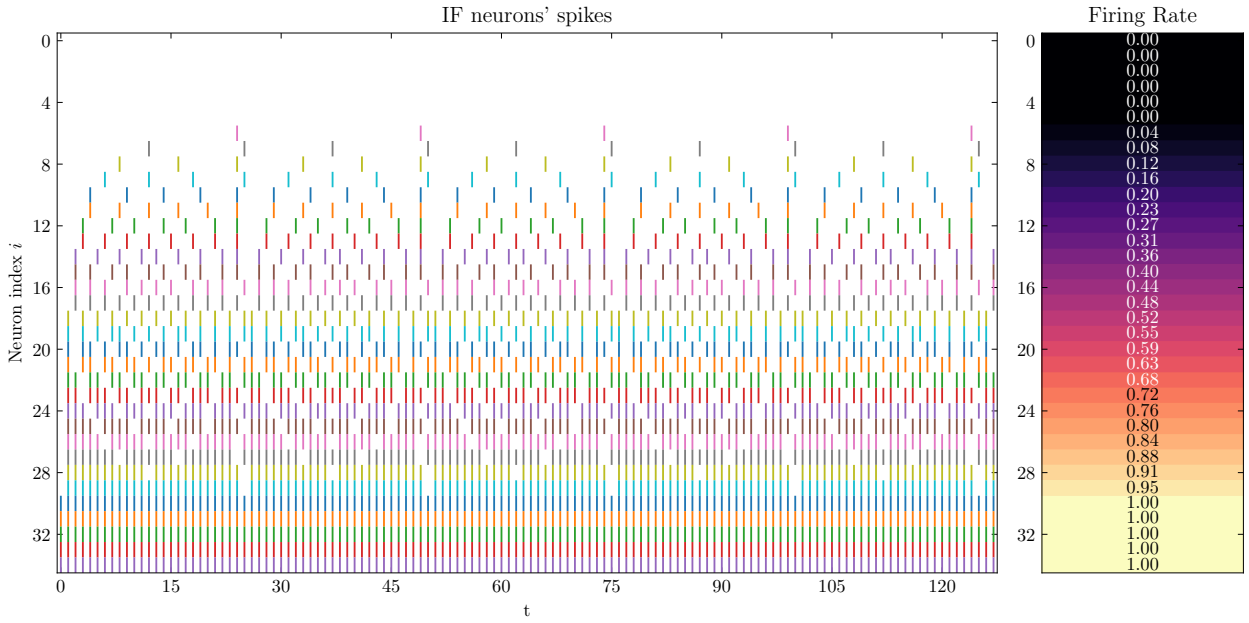
我们给与恒定输入到 IF 神经元，观察其输出脉冲和脉冲发放频率。首先导入相关的模块，新建 IF 神经元层，确定输入并画出每个 IF 神经元的输入 x_i：

```
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)
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：

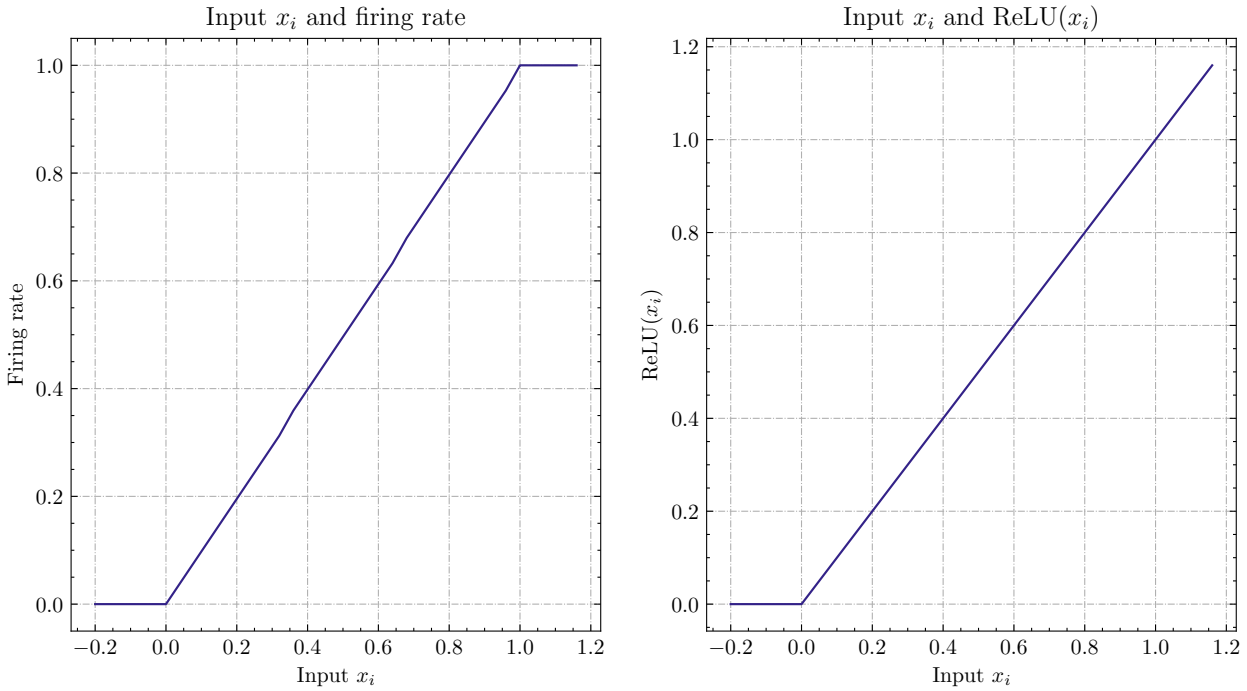
```
s_list = []
for t in range(T):
    s_list.append(if_node(x).unsqueeze(0))

out_spikes = np.asarray(torch.cat(s_list))
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：

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

[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.

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[Page 55, 1] 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[Page 55, 1] 。ann2snn 中的方法也主要来自文献[Page 55, 1]

转换到脉冲神经网络

转换主要解决两个问题:

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，可以将 BN 的参数吸收到前面的参数层中（Linear、Conv2d）
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。

转换中，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[Page 55, 1] 。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。

实现与可选配置

ann2snn 框架在 2022 年 4 月又迎来一次较大更新。取消了 parser 和 simulator 两大类。使用 converter 类替代了之前的方案。目前的方案更加简洁，并且具有更多转换设置空间。

◆ Converter 类该类用于将 ReLU 的 ANN 转换为 SNN。这里实现了常见的三种模式。最常见的是最大电流转换模式，它利用前后层的激活上限，使发放率最高的情况能够对应激活取得最大值的情况。使用这种模式需要将参数 mode 设置为 “max“[#f2]_。99.9% 电流转换模式利用 99.9% 的激活分位点限制了激活上限。使用这种模式需要将参数 mode 设置为 “99.9%“[#f1]_。缩放转换模式下，用户需要给定缩放参数到模式中，即可利用缩放后的激活最大值对电流进行限制。使用这种模式需要将参数 mode 设置为 0-1 的浮点数。

1.7.2 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（见 “ann2snn.sample_models.mnist_cnn“）：

```
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),
```

(续下页)

(接上页)

```
            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：

```
torch.random.manual_seed(0)
torch.cuda.manual_seed(0)
device = 'cuda'
dataset_dir = 'G:/Dataset/mnist'
batch_size = 100
T = 50
```

这里的 T 就是一会儿推理时使用的推理时间步。

如果您想训练的话，还需要初始化数据加载器、优化器、损失函数，例如：

```
lr = 1e-3
epochs = 10
# 定义损失函数
loss_function = nn.CrossEntropyLoss()
# 使用Adam优化器
optimizer = torch.optim.Adam(ann.parameters(), lr=lr, weight_decay=5e-4)
```

训练 ANN。示例中，我们的模型训练了 10 个 epoch。训练时测试集准确率变化情况如下：

```
Epoch: 0 100%|██████████| 600/600 [00:05<00:00, 112.04it/s]
Validating Accuracy: 0.972
Epoch: 1 100%|██████████| 600/600 [00:05<00:00, 105.43it/s]
Validating Accuracy: 0.986
Epoch: 2 100%|██████████| 600/600 [00:05<00:00, 107.49it/s]
Validating Accuracy: 0.987
Epoch: 3 100%|██████████| 600/600 [00:05<00:00, 109.26it/s]
Validating Accuracy: 0.990
Epoch: 4 100%|██████████| 600/600 [00:05<00:00, 103.98it/s]
Validating Accuracy: 0.984
Epoch: 5 100%|██████████| 600/600 [00:05<00:00, 100.42it/s]
Validating Accuracy: 0.989
Epoch: 6 100%|██████████| 600/600 [00:06<00:00, 96.24it/s]
```

(续下页)

(接上页)

```
Validating Accuracy: 0.991
Epoch: 7 100%|██████████| 600/600 [00:05<00:00, 104.97it/s]
Validating Accuracy: 0.992
Epoch: 8 100%|██████████| 600/600 [00:05<00:00, 106.45it/s]
Validating Accuracy: 0.991
Epoch: 9 100%|██████████| 600/600 [00:05<00:00, 111.93it/s]
Validating Accuracy: 0.991
```

训练好模型后，我们快速加载一下模型测试一下保存好的模型性能：

```
model.load_state_dict(torch.load('SJ-mnist-cnn_model-sample.pth'))
acc = val(model, device, test_data_loader)
print('ANN Validating Accuracy: %.4f' % (acc))
```

输出结果如下：

```
100%|██████████| 200/200 [00:02<00:00, 89.44it/s]
ANN Validating Accuracy: 0.9870
```

使用 Converter 进行转换非常简单，只需要参数中设置希望使用的模式即可。例如使用 MaxNorm，需要先定义一个 “ann2snn.Converter“，并且把模型 forward 给这个对象：

```
model_converter = ann2snn.Converter(mode='max', dataloader=train_data_loader)
snn_model = model_converter(model)
```

snn_model 就是输出来的 SNN 模型。

按照这个例子，我们分别定义模式为 “max“，99.9%，1.0/2，1.0/3，1.0/4，“1.0/5“情况下的 SNN 转换并分别推理 T 步得到准确率。

```
print('-------------------------------------------')
print('Converting using MaxNorm')
model_converter = ann2snn.Converter(mode='max', dataloader=train_data_loader)
snn_model = model_converter(model)
print('Simulating...')
mode_max_accs = val(snn_model, device, test_data_loader, T=T)
print('SNN accuracy (simulation %d time-steps): %.4f' % (T, mode_max_accs[-1]))

print('-------------------------------------------')
print('Converting using RobustNorm')
model_converter = ann2snn.Converter(mode='99.9%', dataloader=train_data_loader)
snn_model = model_converter(model)
print('Simulating...')
mode_robust_accs = val(snn_model, device, test_data_loader, T=T)
```

(续下页)

(接上页)

```
print('SNN accuracy (simulation %d time-steps): %.4f' % (T, mode_robust_accs[-1]))

print('-------------------------------------------')
print('Converting using 1/2 max(activation) as scales...')
model_converter = ann2snn.Converter(mode=1.0 / 2, dataloader=train_data_loader)
snn_model = model_converter(model)
print('Simulating...')
mode_two_accs = val(snn_model, device, test_data_loader, T=T)
print('SNN accuracy (simulation %d time-steps): %.4f' % (T, mode_two_accs[-1]))

print('-------------------------------------------')
print('Converting using 1/3 max(activation) as scales')
model_converter = ann2snn.Converter(mode=1.0 / 3, dataloader=train_data_loader)
snn_model = model_converter(model)
print('Simulating...')
mode_three_accs = val(snn_model, device, test_data_loader, T=T)
print('SNN accuracy (simulation %d time-steps): %.4f' % (T, mode_three_accs[-1]))

print('-------------------------------------------')
print('Converting using 1/4 max(activation) as scales')
model_converter = ann2snn.Converter(mode=1.0 / 4, dataloader=train_data_loader)
snn_model = model_converter(model)
print('Simulating...')
mode_four_accs = val(snn_model, device, test_data_loader, T=T)
print('SNN accuracy (simulation %d time-steps): %.4f' % (T, mode_four_accs[-1]))

print('-------------------------------------------')
print('Converting using 1/5 max(activation) as scales')
model_converter = ann2snn.Converter(mode=1.0 / 5, dataloader=train_data_loader)
snn_model = model_converter(model)
print('Simulating...')
mode_five_accs = val(snn_model, device, test_data_loader, T=T)
print('SNN accuracy (simulation %d time-steps): %.4f' % (T, mode_five_accs[-1]))
```

观察控制栏输出：

```
-------------------------------------------
Converting using MaxNorm
100%|██████████| 600/600 [00:04<00:00, 128.25it/s] Simulating...
100%|██████████| 200/200 [00:13<00:00, 14.44it/s] SNN accuracy (simulation 50 time-
↪steps): 0.9777
-------------------------------------------
Converting using RobustNorm
100%|██████████| 600/600 [00:19<00:00, 31.06it/s] Simulating...
```

(续下页)

(接上页)

```
100%|██████████| 200/200 [00:13<00:00, 14.75it/s] SNN accuracy (simulation 50 time-
↪steps): 0.9841
---------------------------------------------
Converting using 1/2 max(activation) as scales...
100%|██████████| 600/600 [00:04<00:00, 126.64it/s] ]Simulating...
100%|██████████| 200/200 [00:13<00:00, 14.90it/s] SNN accuracy (simulation 50 time-
↪steps): 0.9844
---------------------------------------------
Converting using 1/3 max(activation) as scales
100%|██████████| 600/600 [00:04<00:00, 126.27it/s] Simulating...
100%|██████████| 200/200 [00:13<00:00, 14.73it/s] SNN accuracy (simulation 50 time-
↪steps): 0.9828
---------------------------------------------
Converting using 1/4 max(activation) as scales
100%|██████████| 600/600 [00:04<00:00, 128.94it/s] Simulating...
100%|██████████| 200/200 [00:13<00:00, 14.47it/s] SNN accuracy (simulation 50 time-
↪steps): 0.9747
---------------------------------------------
Converting using 1/5 max(activation) as scales
100%|██████████| 600/600 [00:04<00:00, 121.18it/s] Simulating...
100%|██████████| 200/200 [00:13<00:00, 14.42it/s] SNN accuracy (simulation 50 time-
↪steps): 0.9487
---------------------------------------------
```

模型转换的速度可以看到是非常快的。模型推理速度 200 步仅需 11s 完成（GTX 2080ti）。根据模型输出的随时间变化的准确率，我们可以绘制不同设置下的准确率图像。

```
fig = plt.figure()
plt.plot(np.arange(0, T), mode_max_accs, label='mode: max')
plt.plot(np.arange(0, T), mode_robust_accs, label='mode: 99.9%')
plt.plot(np.arange(0, T), mode_two_accs, label='mode: 1.0/2')
plt.plot(np.arange(0, T), mode_three_accs, label='mode: 1.0/3')
plt.plot(np.arange(0, T), mode_four_accs, label='mode: 1.0/4')
plt.plot(np.arange(0, T), mode_five_accs, label='mode: 1.0/5')
plt.legend()
plt.xlabel('t')
plt.ylabel('Acc')
plt.show()
```

不同的设置可以得到不同的结果，有的推理速度快，但是最终精度低，有的推理慢，但是精度高。用户可以根据自己的需求选择模型设置。

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):
    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)
```

(续下页)

(接上页)

```
        )

        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
```

(续下页)

(接上页)

```
        # 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)
```

(续下页)

(接上页)

```
            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():
            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：

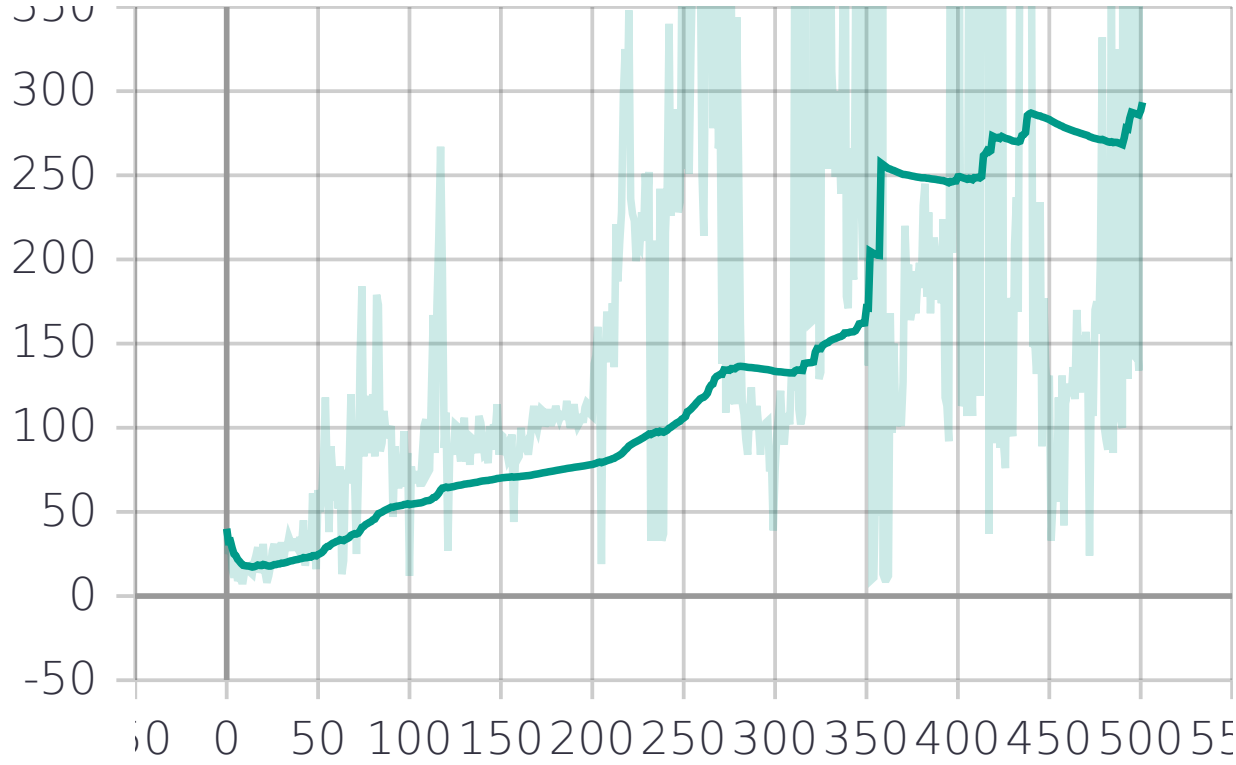
```
>>> from spikingjelly.clock_driven.examples import Spiking_DQN_state
>>> Spiking_DQN_state.play(use_cuda=False, pt_path='./model/CartPole-v0/policy_net_
↪256_max.pt', env_name='CartPole-v0', hidden_size=256, played_frames=300)
```

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：

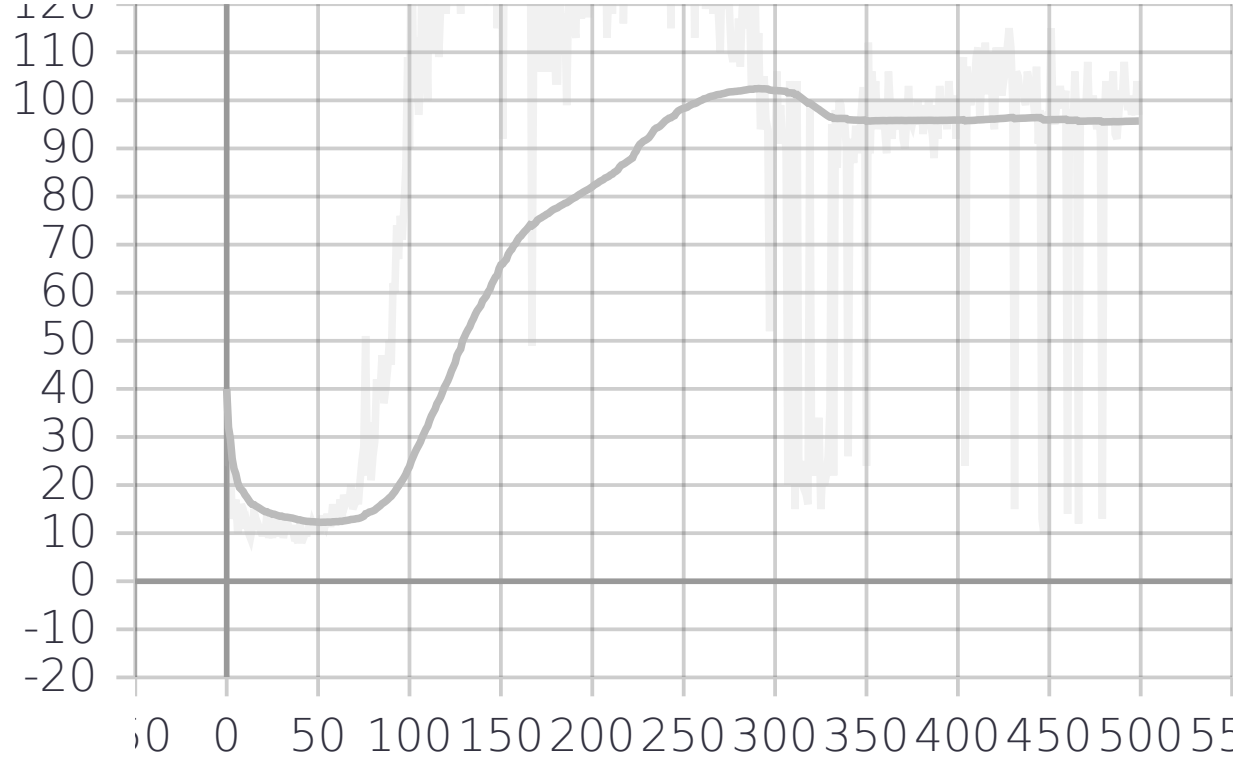
训练 16 次的效果：

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):
```

(续下页)

(接上页)

```
            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()
```

(续下页)

(接上页)

```
            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()
```

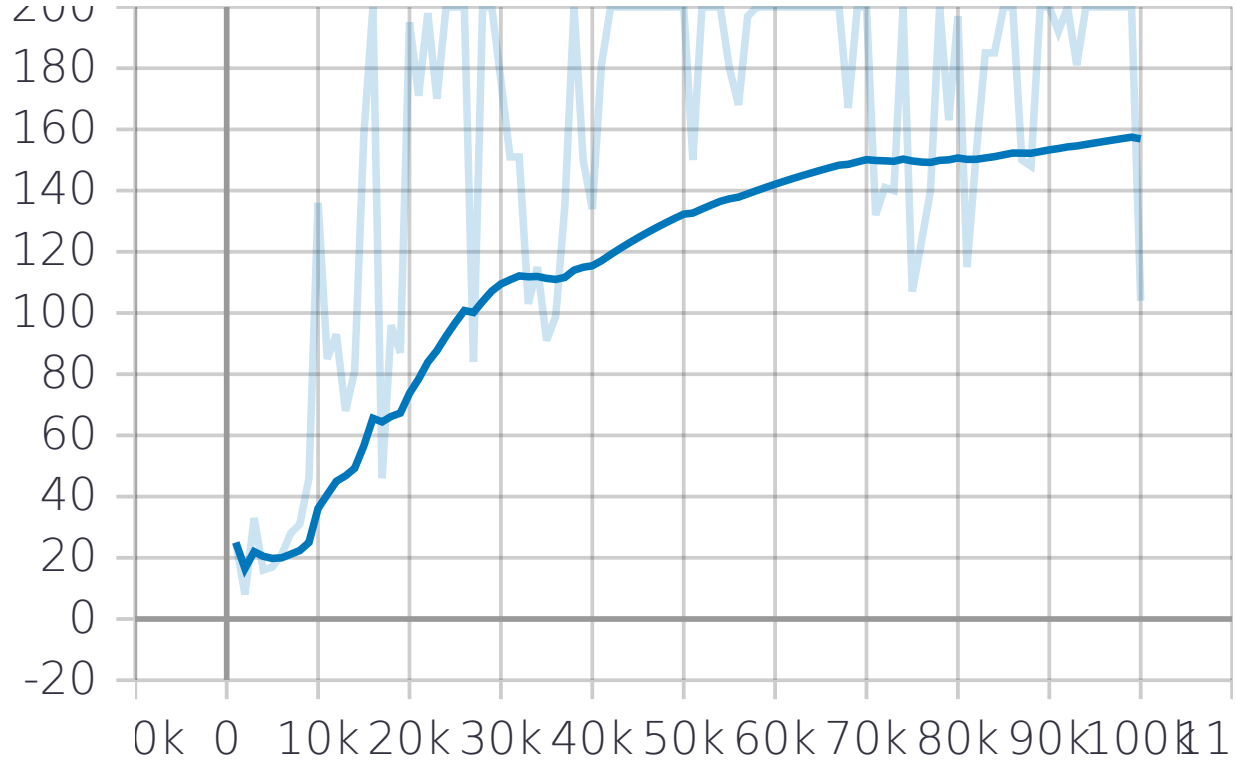
另外一个需要注意的地方是，SNN 是有状态的，因此每次前向传播后，不要忘了将网络 `reset`。

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：

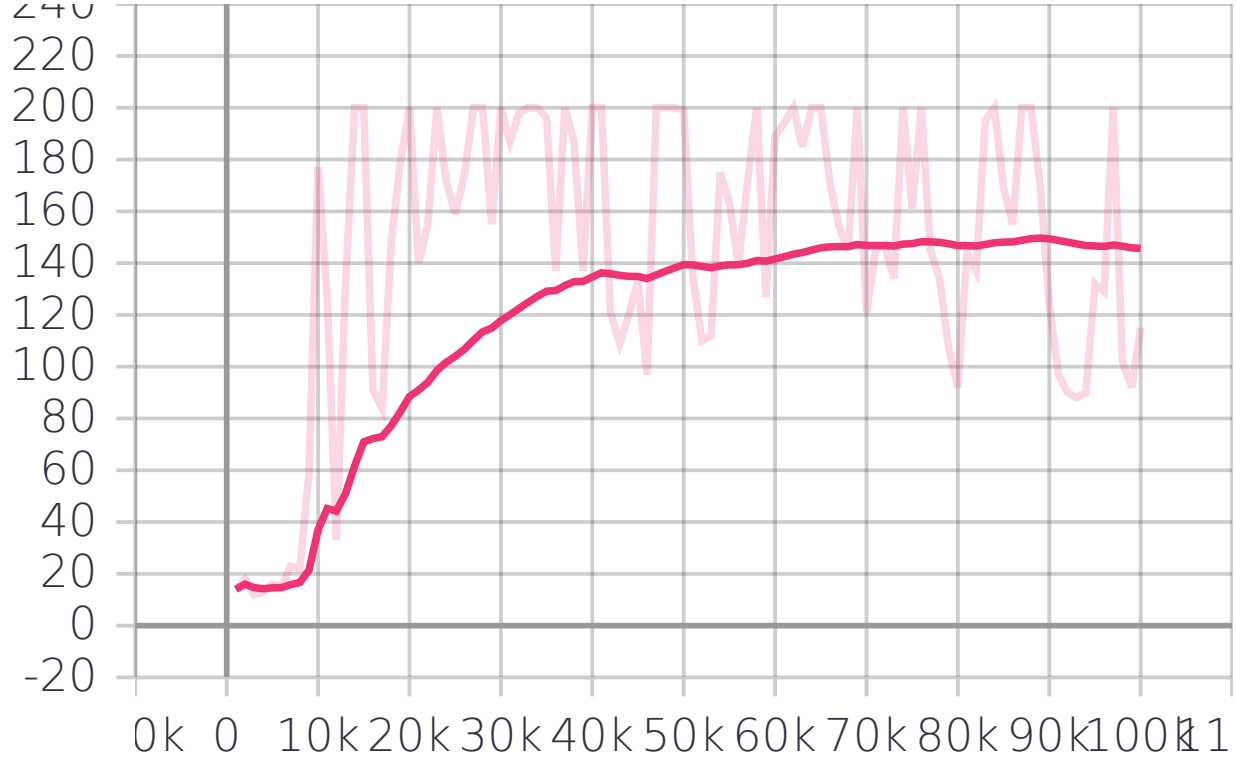
```
>>> python Spiking_A2C.py
```

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
```

(续下页)

(接上页)

```
    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):
```

(续下页)

(接上页)

```
            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))
```

(续下页)

(接上页)

```
        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)
```

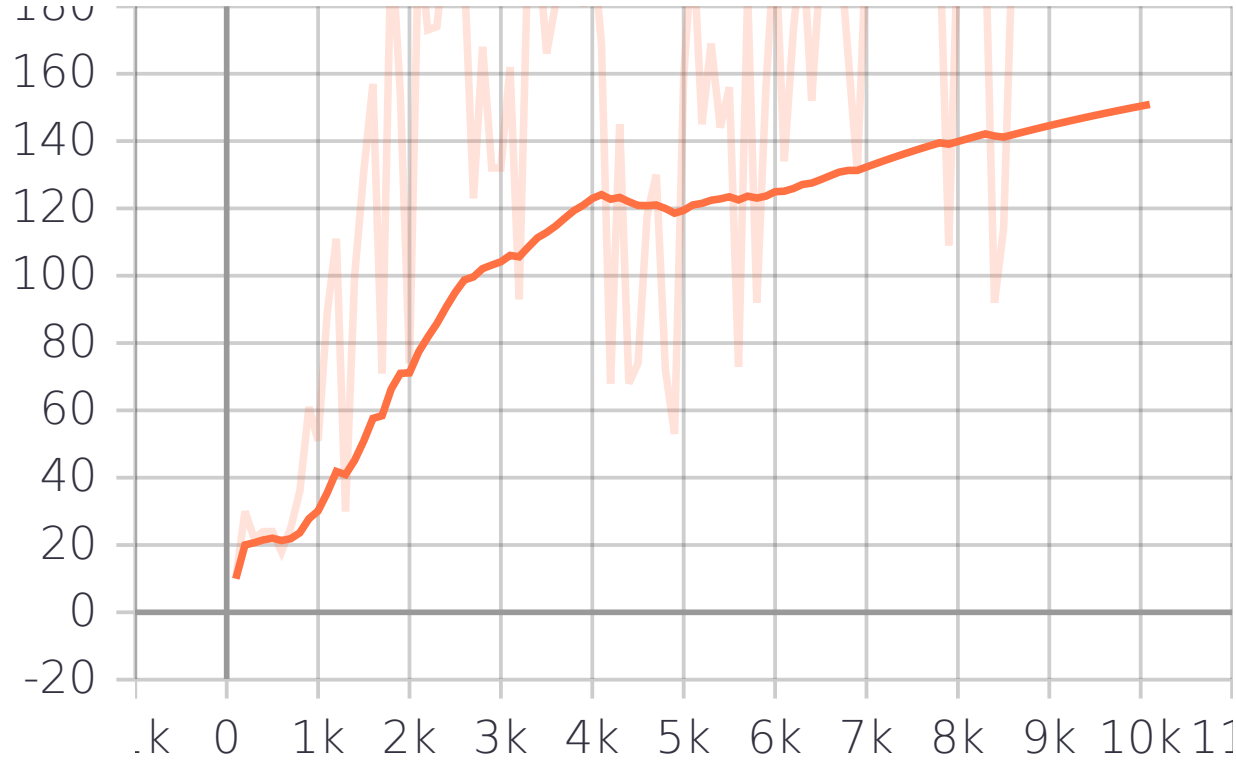
另外一个需要注意的地方是，SNN 是有状态的，因此每次前向传播后，不要忘了将网络 `reset`。

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：

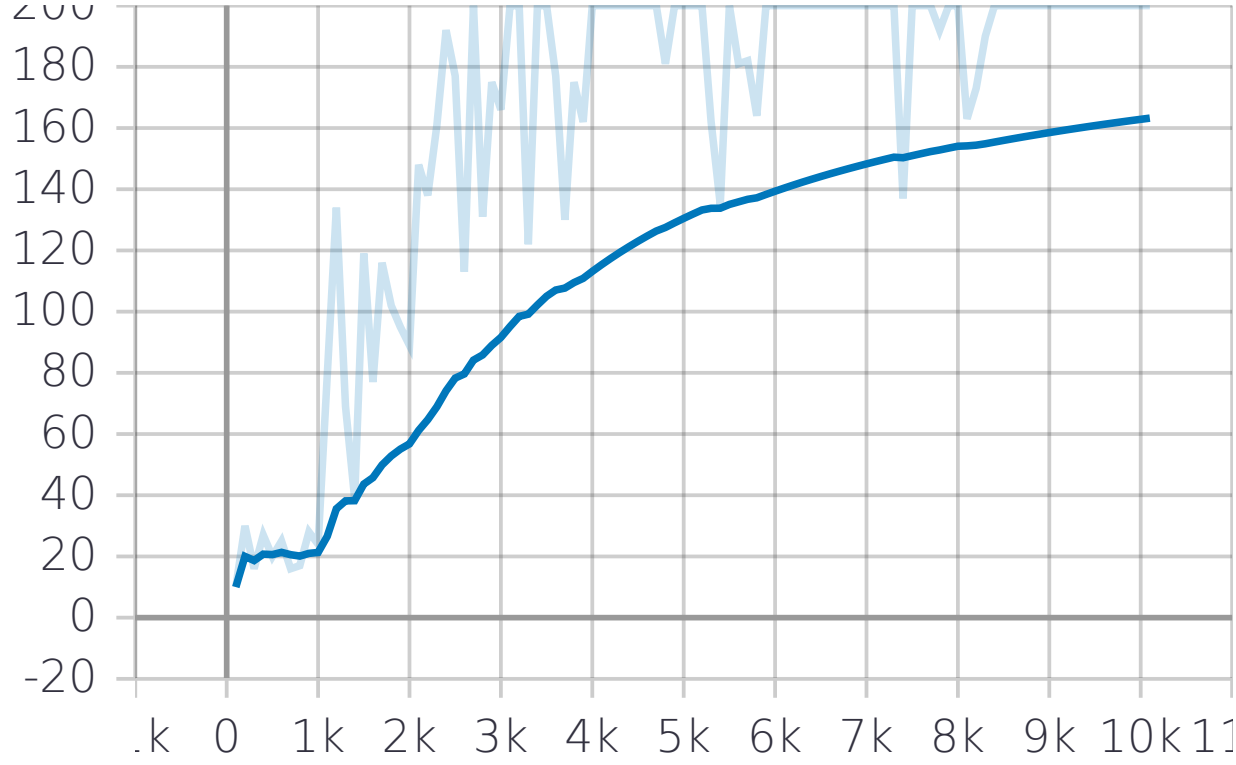
```
>>> python Spiking_PPO.py
```

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)
```

(续下页)

(接上页)

```
    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)
```

(续下页)

(接上页)

```
    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)
```

(续下页)

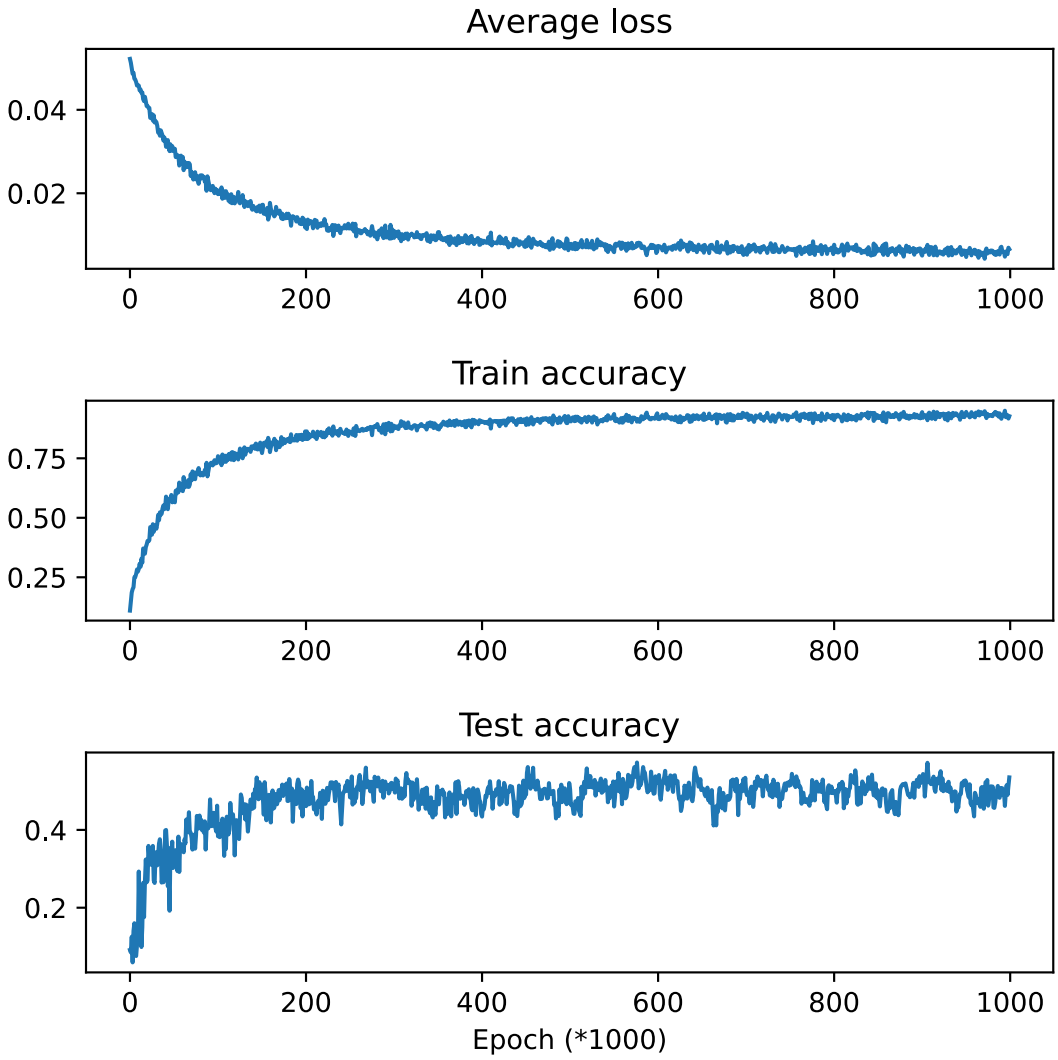
(接上页)

```
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')
```

(续下页)

(接上页)

```
# 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？
```

(续下页)

(接上页)

```
    # 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：

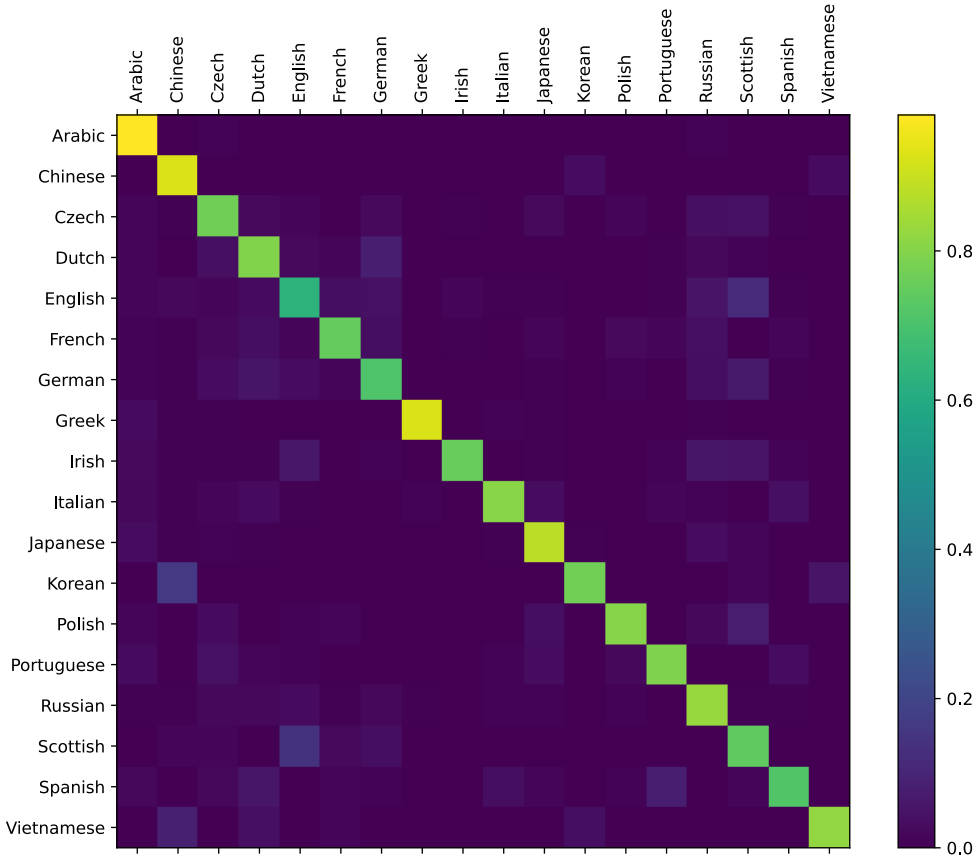
```
Testing...
Calculating testing accuracy...
Test accuracy: 0.512, Random guess: 0.056
请输入一个姓氏以判断其属于哪种语系:> YU
> YU
(0.18) Scottish
(0.12) English
(0.11) Italian
请输入一个姓氏以判断其属于哪种语系:> Yu
> Yu
(0.63) Chinese
(0.23) Korean
(0.07) Vietnamese
请输入一个姓氏以判断其属于哪种语系:> Zou
> Zou
(1.00) Chinese
(0.00) Arabic
(0.00)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

(续下页)

(接上页)

```
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.Sequential):
    def __init__(self, *args):
        super().__init__(*args)

    def forward(self, x_seq: torch.Tensor):
        """
        :param x_seq: shape=[T, batch_size, ...]
        :type x_seq: torch.Tensor
        :return: y_seq, shape=[T, batch_size, ...]
        :rtype: torch.Tensor
        """
        y_seq = []
        for t in range(x_seq.shape[0]):
            y_seq.append(super().forward(x_seq[t]))

        for t in range(y_seq.__len__()):
            y_seq[t] = y_seq[t].unsqueeze(0)
        return torch.cat(y_seq, 0)
```

我们使用这种方式来包装一个 IF 神经元：

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

neuron_num = 4
T = 8
if_node = neuron.IFNode()
x = torch.rand([T, neuron_num]) * 2
for t in range(T):
```

(续下页)

(接上页)

```
    print(f'if_node output spikes at t={t}', if_node(x[t]))
functional.reset_net(if_node)

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

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

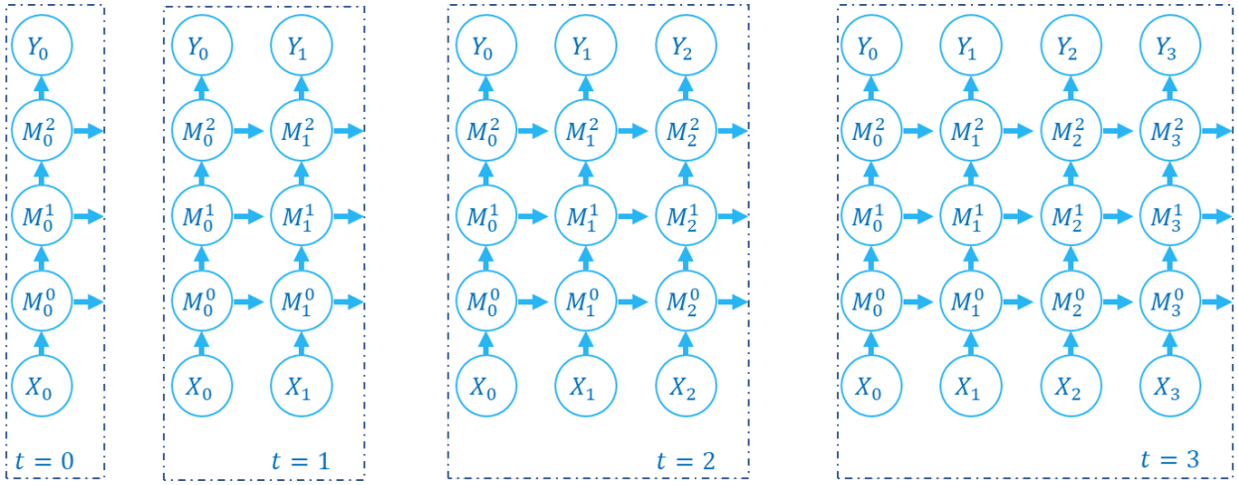
在以往的教程和样例中，我们定义的网络在运行时，是按照 *逐步传播 (step-by-step)* 的方式，例如上文中的：

```
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])
```

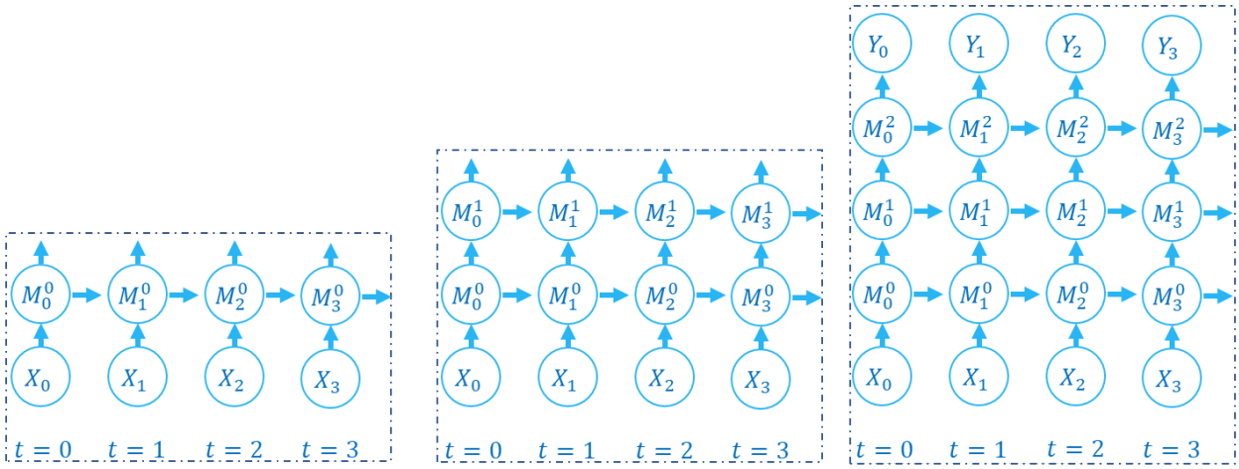
前向传播的计算图的构建顺序如下所示：



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。

1.12.3 包装前向传播

使用 SeqToANNContainer 对无状态的 ANN 层进行包装后，网络的 state_dict 中层的名字 .keys() 会发生变化，因为我们额外引入了一个包装器。例如：

```
net_step_by_step = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=3, padding=1, bias=False),
    nn.BatchNorm2d(16),
    neuron.IFNode()
)

net_layer_by_layer = nn.Sequential(
    layer.SeqToANNContainer(
        nn.Conv2d(3, 16, kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(16),
    ),
    neuron.MultiStepIFNode()
```

(续下页)

(接上页)

```
)

print('net_step_by_step.state_dict:', net_step_by_step.state_dict().keys())
print('net_layer_by_layer.state_dict:', net_layer_by_layer.state_dict().keys())
```

输出为：

```
net_step_by_step.state_dict: odict_keys(['0.weight', '1.weight', '1.bias', '1.running_
↪mean', '1.running_var', '1.num_batches_tracked'])
net_layer_by_layer.state_dict: odict_keys(['0.0.weight', '0.1.weight', '0.1.bias', '0.
↪1.running_mean', '0.1.running_var', '0.1.num_batches_tracked'])
```

名称不一样，会给加载模型权重带来麻烦。例如，我们想构建一个多步版本的 Spiking ResNet-18 (*spikingjelly.clock_driven.model.spiking_resnet.spiking_resnet18*)，且希望这个网络能够加载 ANN 的预训练模型权重。直接使用 `SeqToANNContainer` 构建出的网络，`state_dict` 与 ANN 的并不相同，无法直接加载。为了避免这种问题，我们可以不使用 `SeqToANNContainer` 对 ANN 层包装，而是转为包装 ANN 层的前向传播代码。下面是示例代码：

```
class NetStepByStep(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(3, 16, kernel_size=3, padding=1, bias=False)
        self.bn = nn.BatchNorm2d(16)
        self.sn = neuron.IFNode()

    def forward(self, x):
        # x.shape = [N, C, H, W]
        x = self.conv(x)
        x = self.bn(x)
        x = self.sn(x)
        return x


class NetLayerByLayer1(NetStepByStep):

    def forward(self, x_seq):
        # x_seq.shape = [T, N, C, H, W]
        x_seq = functional.seq_to_ann_forward(x_seq, [self.conv, self.bn])
        x_seq = functional.multi_step_forward(x_seq, self.sn)
        return x_seq


class NetLayerByLayer2(NetStepByStep):
```

(续下页)

(接上页)

```
    def __init__(self):
        super().__init__()

        # replace single-step neuron to multi-step neuron
        del self.sn
        self.sn = neuron.MultiStepIFNode()

    def forward(self, x_seq):
        # x_seq.shape = [T, N, C, H, W]
        x_seq = functional.seq_to_ann_forward(x_seq, [self.conv, self.bn])
        x_seq = self.sn(x_seq)
        return x_seq
```

NetStepByStep, NetLayerByLayer1, NetLayerByLayer2 的 state_dict.keys() 完全相同的，模型权重可以互相加载。

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

在*spikingjelly.clock_driven.neuron*提供了多步版本的神经元。与单步版本相比，多步神经元增加了 cupy 后端。cupy 后端将各种运算都封装到了一个 CUDA 内核，因此速度比默认 pytorch 后端更快。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.clock_driven import neuron, surrogate, cu_kernel_opt
import torch


def cal_forward_t(multi_step_neuron, x, repeat_times):
    with torch.no_grad():
        used_t = cu_kernel_opt.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_()
```

(续下页)

(接上页)

```
def cal_forward_backward_t(multi_step_neuron, x, repeat_times):
    x.requires_grad_(True)
    used_t = cu_kernel_opt.cal_fun_t(repeat_times, x.device, forward_backward, multi_
↪step_neuron, x)
    return used_t * 1000


device = 'cuda:0'
repeat_times = 1024
ms_lif = neuron.MultiStepLIFNode(surrogate_function=surrogate.ATan(alpha=2.0))


ms_lif.to(device)
N = 2 ** 20
print('forward')
ms_lif.eval()
for T in [8, 16, 32, 64, 128]:
    x = torch.rand(T, N, device=device)
    ms_lif.backend = 'torch'
    print(T, cal_forward_t(ms_lif, x, repeat_times), end=', ')
    ms_lif.backend = 'cupy'
    print(cal_forward_t(ms_lif, x, repeat_times))

print('forward and backward')
ms_lif.train()
for T in [8, 16, 32, 64, 128]:
    x = torch.rand(T, N, device=device)
    ms_lif.backend = 'torch'
    print(T, cal_forward_backward_t(ms_lif, x, repeat_times), end=', ')
    ms_lif.backend = 'cupy'
    print(cal_forward_backward_t(ms_lif, x, repeat_times))
```

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：

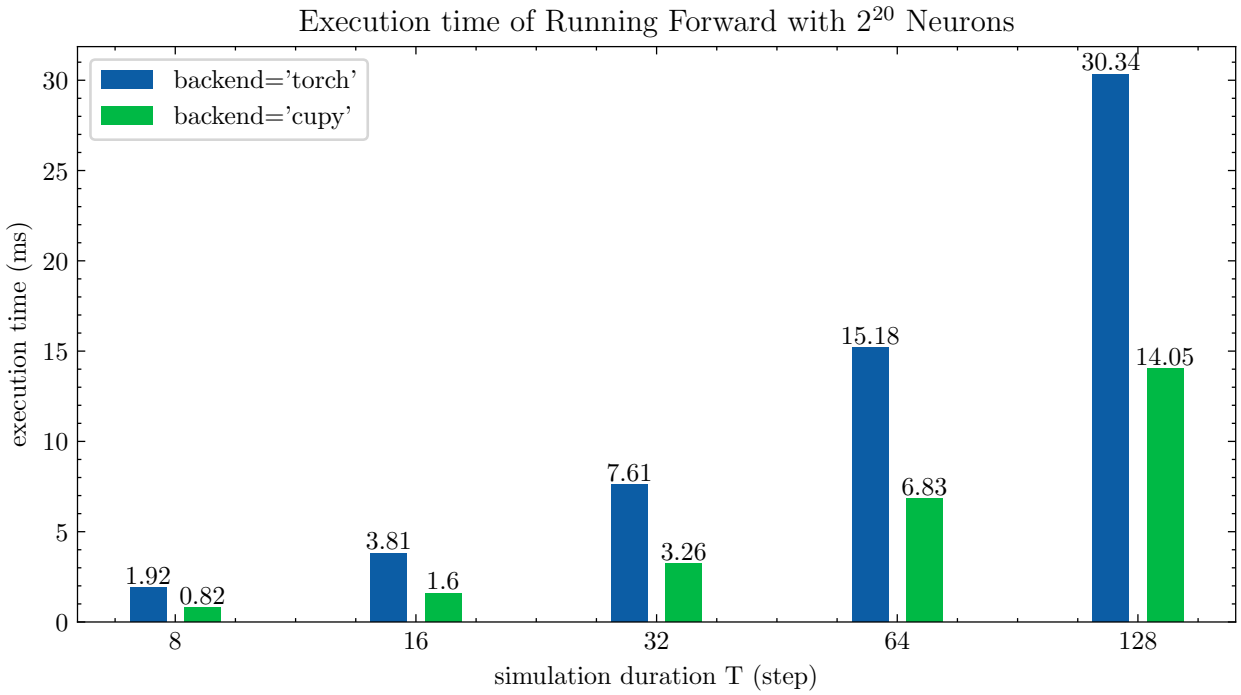
```
forward
8 1.9180845527841939, 0.8166529733273364
16 3.8143536958727964, 1.6002442711169351
32 7.6071328955436, 3.2570467449772877
64 15.181676714490777, 6.82808195671214
128 30.344632044631226, 14.053565065751172
```

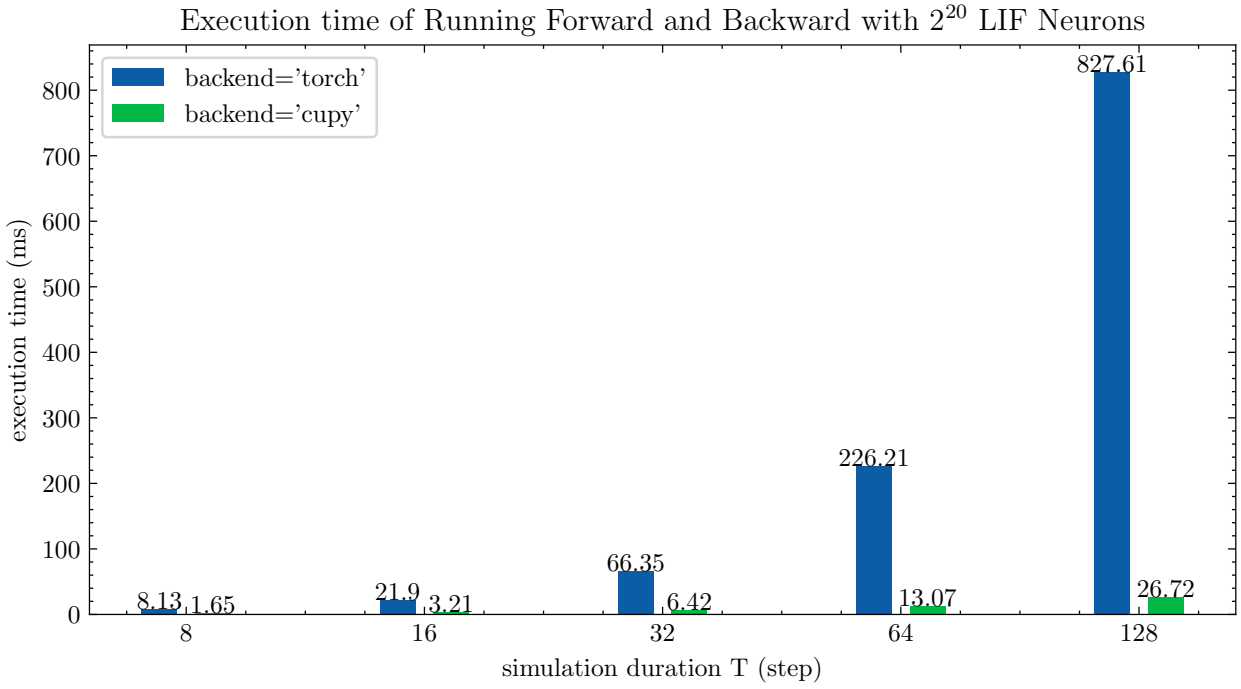
(续下页)

(接上页)

```
forward and backward
8 8.131792200288146, 1.6501817200662572
16 21.89934094545265, 3.210343387223702
32 66.34630815216269, 6.41730432241161
64 226.20835550819152, 13.073845567419085
128 827.6064751953811, 26.71502177403795
```

将结果画成柱状图：





可以发现，使用 cupy 后端速度明显快于原生 pytorch 后端。

1.13.2 加速深度脉冲神经网络

现在让我们使用多步和 cupy 后端神经元，重新实现*时间驱动：使用卷积 SNN 识别 Fashion-MNIST* 中的网络。我们只需要更改一下网络结构，无需进行其他的改动：

```
class CupyNet(nn.Module):
    def __init__(self, T):
        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.MultiStepIFNode(surrogate_function=surrogate.ATan(), backend='cupy
↪'),
            layer.SeqToANNContainer(
                    nn.MaxPool2d(2, 2),  # 14 * 14
                    nn.Conv2d(128, 128, kernel_size=3, padding=1, bias=False),
                    nn.BatchNorm2d(128),
            ),
```

(续下页)

(接上页)

```
            neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), backend='cupy
↪'),
            layer.SeqToANNContainer(
                nn.MaxPool2d(2, 2),  # 7 * 7
                nn.Flatten(),
            ),
        )
        self.fc = nn.Sequential(
            layer.SeqToANNContainer(nn.Linear(128 * 7 * 7, 128 * 4 * 4, bias=False)),
            neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), backend='cupy
↪'),
            layer.SeqToANNContainer(nn.Linear(128 * 4 * 4, 10, bias=False)),
            neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), backend='cupy
↪'),
        )


    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]

        return self.fc(self.conv(x_seq)).mean(0)
```

完整的代码可见于*spikingjelly.clock_driven.examples.conv_fashion_mnist*。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：

```
(pytorch-env) root@e8b6e4800dae4011eb0918702bd7ddedd51c-fangw1598-0:/# python -m␣
↪spikingjelly.clock_driven.examples.conv_fashion_mnist -opt SGD -data_dir /userhome/
↪datasets/FashionMNIST/ -amp -cupy

Namespace(T=4, T_max=64, amp=True, b=128, cupy=True, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
CupyNet(
  (static_conv): Sequential(
    (0): Conv2d(1, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
  )
  (conv): Sequential(
    (0): MultiStepIFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
```

(续下页)

(接上页)

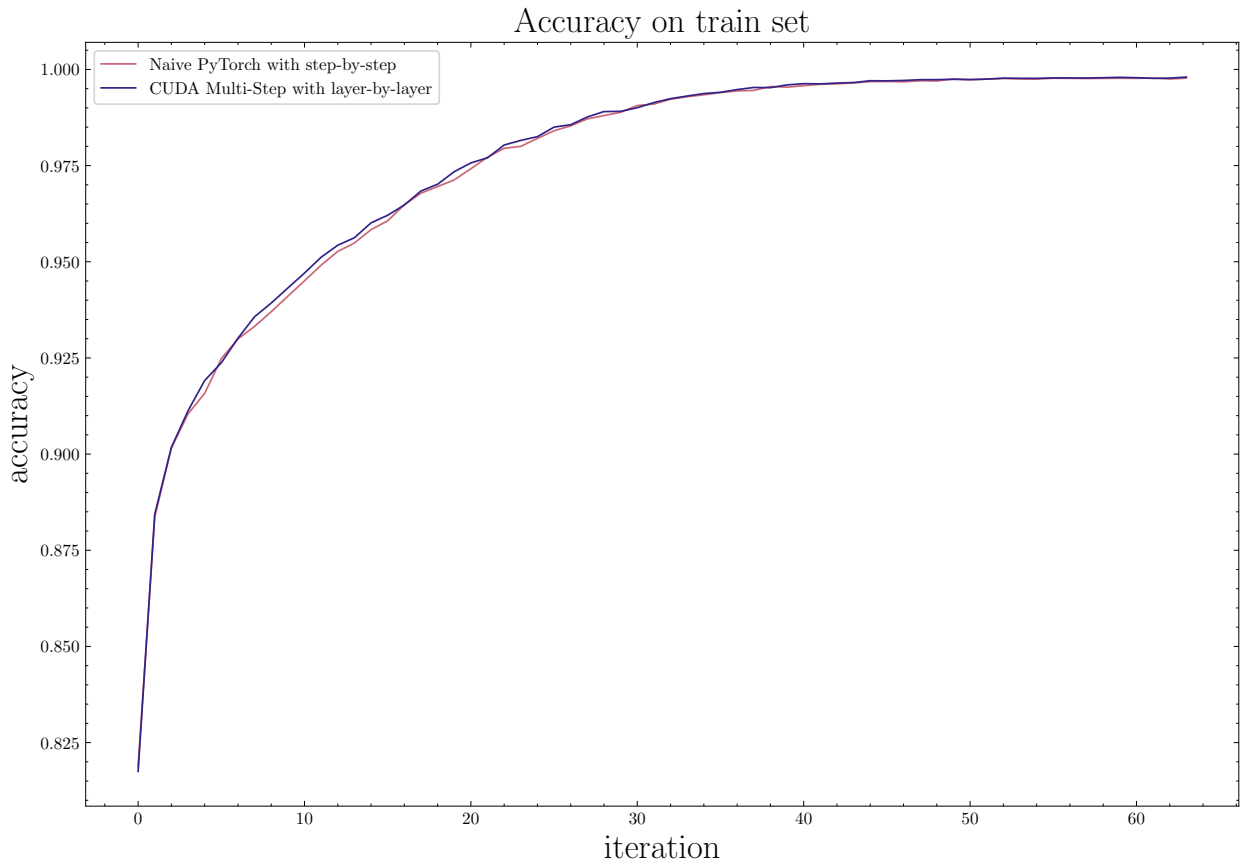
```
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (1): SeqToANNContainer(
      (module): Sequential(
        (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
        (1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
        (2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (2): MultiStepIFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (3): SeqToANNContainer(
      (module): Sequential(
        (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
        (1): Flatten(start_dim=1, end_dim=-1)
      )
    )
  )
  (fc): Sequential(
    (0): SeqToANNContainer(
      (module): Linear(in_features=6272, out_features=2048, bias=False)
    )
    (1): MultiStepIFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (2): SeqToANNContainer(
      (module): Linear(in_features=2048, out_features=10, bias=False)
    )
    (3): MultiStepIFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
  )
)
Mkdir ./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp_cupy.
Namespace(T=4, T_max=64, amp=True, b=128, cupy=True, data_dir='/userhome/datasets/
```

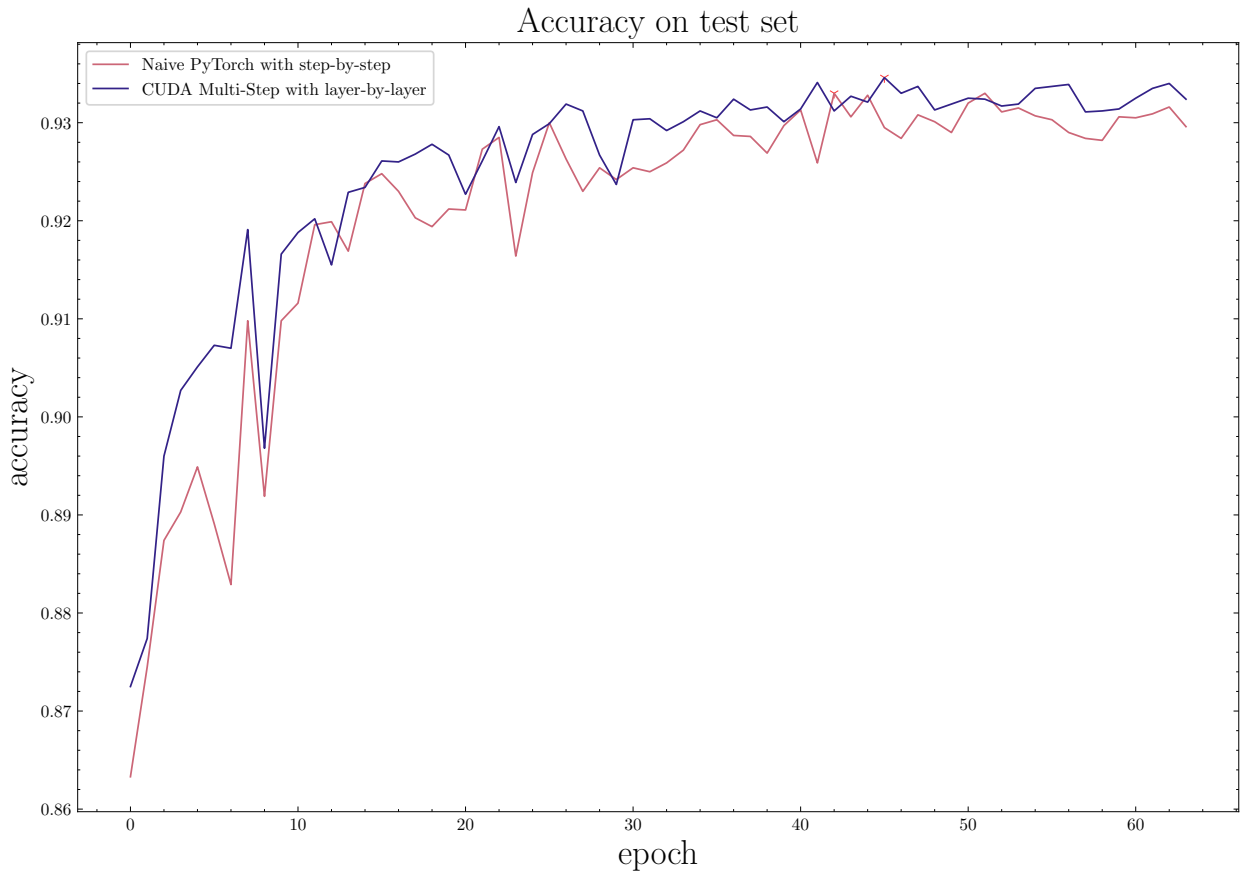
(续下页)

(接上页)

```
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp_cupy
epoch=0, train_loss=0.028574782584865507, train_acc=0.8175080128205128, test_loss=0.
↪020883125430345536, test_acc=0.8725, max_test_acc=0.8725, total_time=13.
↪037598133087158
Namespace(T=4, T_max=64, amp=True, b=128, cupy=True, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp_cupy

...

epoch=62, train_loss=0.001055751721853287, train_acc=0.9977463942307693, test_loss=0.
↪010815625159442425, test_acc=0.934, max_test_acc=0.9346, total_time=11.
↪059867858886719
Namespace(T=4, T_max=64, amp=True, b=128, cupy=True, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp_cupy
epoch=63, train_loss=0.0010632637413514631, train_acc=0.9980134882478633, test_loss=0.
↪010720000202953816, test_acc=0.9324, max_test_acc=0.9346, total_time=11.
↪128222703933716
```

最终的正确率是 93.46%，与使用 *CUDA* 增强的神经元与逐层传播进行加速 中的 93.3% 相差无几，两者在训练过程中的测试集正确率曲线如下：


Accuracy on train set
Naive PyTorch with step-by-step
CUDA Multi-Step with layer-by-layer
accuracy
iteration
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0
10
20
30
40
50
60



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，一个 epoch 的耗时为原始网络的 69%，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(
```

(续下页)

(接上页)

```
                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 $X \sim \mathcal{U}(1,3)$

```
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], N-Caltech101[Page 104, 1], ASLDVS[4] 等。所有数据集的处理都遵循类似的步骤，开发人员也可以很轻松的添加新数据集代码。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 自动下载和手动下载

CIFAR10-DVS 等数据集支持自动下载。支持自动下载的数据集，在首次运行时原始数据集将会被下载到数据集根目录下的 `download` 文件夹。每个数据集的 `downloadable()` 函数定义了该数据集是否能够自动下载，而 `resource_url_md5()` 函数定义了各个文件的下载链接和 MD5。示例：

```
from spikingjelly.datasets.cifar10_dvs import CIFAR10DVS
from spikingjelly.datasets.dvs128_gesture import DVS128Gesture

print('CIFAR10-DVS downloadable', CIFAR10DVS.downloadable())
print('resource, url, md5/n', CIFAR10DVS.resource_url_md5())

print('DVS128Gesture downloadable', DVS128Gesture.downloadable())
print('resource, url, md5/n', DVS128Gesture.resource_url_md5())
```

输出为：

```
CIFAR10-DVS downloadable True
resource, url, md5
 [('airplane.zip', 'https://ndownloader.figshare.com/files/7712788',
↪'0afd5c4bf9ae06af762a77b180354fdd'), ('automobile.zip', 'https://ndownloader.
↪figshare.com/files/7712791', '8438dfeba3bc970c94962d995b1b9bdd'), ('bird.zip',
↪'https://ndownloader.figshare.com/files/7712794', 'a9c207c91c55b9dc2002dc21c684d785
↪'), ('cat.zip', 'https://ndownloader.figshare.com/files/7712812',
↪'52c63c677c2b15fa5146a8daf4d56687'), ('deer.zip', 'https://ndownloader.figshare.com/
↪files/7712815', 'b6bf21f6c04d21ba4e23fc3e36c8a4a3'), ('dog.zip', 'https://
↪ndownloader.figshare.com/files/7712818', 'f379ebdf6703d16e0a690782e62639c3'), (
```

(续下页)

[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] 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.

(接上页)

```
↪'frog.zip', 'https://ndownloader.figshare.com/files/7712842',
↪'cad6ed91214b1c7388a5f6ee56d08803'), ('horse.zip', 'https://ndownloader.figshare.
↪com/files/7712851', 'e7cbbf77bec584ffbf913f00e682782a'), ('ship.zip', 'https://
↪ndownloader.figshare.com/files/7712836', '41c7bd7d6b251be82557c6cce9a7d5c9'), (
↪'truck.zip', 'https://ndownloader.figshare.com/files/7712839',
↪'89f3922fd147d9aeff89e76a2b0b70a7')]
DVS128Gesture downloadable False
resource, url, md5
 [('DvsGesture.tar.gz', 'https://ibm.ent.box.com/s/3hiq58ww1pbbjrinh367ykfdf60xsfm8/
↪folder/50167556794', '8a5c71fb11e24e5ca5b11866ca6c00a1'), ('gesture_mapping.csv',
↪'https://ibm.ent.box.com/s/3hiq58ww1pbbjrinh367ykfdf60xsfm8/folder/50167556794',
↪'109b2ae64a0e1f3ef535b18ad7367fd1'), ('LICENSE.txt', 'https://ibm.ent.box.com/s/
↪3hiq58ww1pbbjrinh367ykfdf60xsfm8/folder/50167556794',
↪'065e10099753156f18f51941e6e44b66'), ('README.txt', 'https://ibm.ent.box.com/s/
↪3hiq58ww1pbbjrinh367ykfdf60xsfm8/folder/50167556794',
↪'a0663d3b1d8307c329a43d949ee32d19')]
```

DVS128 Gesture 数据集不支持自动下载，但它的 `resource_url_md5()` 函数会打印出获取下载地址的网址。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 `E:/datasets/DVS128Gesture/download`，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

创建训练集和测试集，其中参数 `data_type='event'` 表示我们使用 Event 数据。

```
from spikingjelly.datasets.dvs128_gesture import DVS128Gesture

root_dir = 'D:/datasets/DVS128Gesture'
train_set = DVS128Gesture(root_dir, train=True, data_type='event')
```

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 `extract` 文件夹
2. 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：

```
The [D:/datasets/DVS128Gesture/download] directory for saving downloaed files already␣
↪exists, check files...
Mkdir [D:/datasets/DVS128Gesture/extract].
Extract [D:/datasets/DVS128Gesture/download/DvsGesture.tar.gz] to [D:/datasets/
↪DVS128Gesture/extract].
Mkdir [D:/datasets/DVS128Gesture/events_np].
Start to convert the origin data from [D:/datasets/DVS128Gesture/extract] to [D:/
↪datasets/DVS128Gesture/events_np] in np.ndarray format.
Mkdir [('D:/datasets/DVS128Gesture//events_np//train', 'D:/datasets/DVS128Gesture//
↪events_np//test').
Mkdir ['0', '1', '10', '2', '3', '4', '5', '6', '7', '8', '9'] in [D:/datasets/
↪DVS128Gesture/events_np/train] and ['0', '1', '10', '2', '3', '4', '5', '6', '7', '8
↪', '9'] in [D:/datasets/DVS128Gesture/events_np/test].
Start the ThreadPoolExecutor with max workers = [8].
Start to split [D:/datasets/DVS128Gesture/extract/DvsGesture/user02_fluorescent.
↪aedat] to samples.
[D:/datasets/DVS128Gesture/events_np/train/0/user02_fluorescent_0.npz] saved.
[D:/datasets/DVS128Gesture/events_np/train/1/user02_fluorescent_0.npz] saved.

......

[D:/datasets/DVS128Gesture/events_np/test/8/user29_lab_0.npz] saved.
[D:/datasets/DVS128Gesture/events_np/test/9/user29_lab_0.npz] saved.
[D:/datasets/DVS128Gesture/events_np/test/10/user29_lab_0.npz] saved.
Used time = [1017.27s].
All aedat files have been split to samples and saved into [('D:/datasets/
↪DVS128Gesture//events_np//train', 'D:/datasets/DVS128Gesture//events_np//test')].
```

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_np 文件夹，其中包含训练集和测试集：

```
|-- events_np
|   |-- test
|   `-- train
```

打印一个数据：

```
event, label = train_set[0]
for k in event.keys():
```

(续下页)

(接上页)

```
    print(k, event[k])
print('label', label)
```

得到输出为：

```
t [80048267 80048277 80048278 ... 85092406 85092538 85092700]
x [49 55 55 ... 60 85 45]
y [82 92 92 ... 96 86 90]
p [1 0 0 ... 1 0 0]
label 0
```

其中 `event` 使用字典格式存储 Events 数据，键为 `['t', 'x', 'y', 'p']`；`label` 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[5] 的实现方式。。我们将原始的 Event 数据记为 $E(x_i, y_i, t_i, p_i), 0 \leq i < N$；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 (p, x, y) 位置的像素值为 $F(j, p, x, y)$；$F(j)$ 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, data_type='frame', frames_number=20,␣
↪split_by='number')
```

命令行的输出为：

```
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/0].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/1].
```

(续下页)

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

(接上页)

```
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/10].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/2].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/3].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/4].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/5].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/6].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/7].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/8].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/9].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/0].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/1].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/10].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/2].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/3].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/4].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/5].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/6].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/7].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/8].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/9].
Start ThreadPoolExecutor with max workers = [8].
Start to integrate [D:/datasets/DVS128Gesture/events_np/test/0/user24_fluorescent_0.
↪npz] to frames and save to [D:/datasets/DVS128Gesture/frames_number_20_split_by_
↪number/test/0].
Start to integrate [D:/datasets/DVS128Gesture/events_np/test/0/user24_fluorescent_led_
↪0.npz] to frames and save to [D:/datasets/DVS128Gesture/frames_number_20_split_by_
↪number/test/0].

......

Frames [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/9/user23_lab_
↪0.npz] saved.Frames [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/
↪train/9/user23_led_0.npz] saved.

Used time = [102.11s].
```

运行后，同级目录下会出现 `frames_number_20_split_by_number` 文件夹，这里存放了积分生成的 Frame 数据。

打印一个数据：

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

得到输出为：

```
(20, 2, 128, 128)
```

查看 1 个积分好的 Frame 数据：

```
from spikingjelly.datasets import play_frame
frame, label = train_set[500]
play_frame(frame)
```

显示效果如下图所示：

1.15.4 固定时间间隔积分

使用固定时间间隔积分，更符合实际物理系统。例如每 10 ms 积分一次，则长度为 L ms 的数据，可以得到 math.floor(L / 10) 帧。但神经形态数据集中每个样本的长度往往不相同，因此会得到不同长度的帧数据。使用惊蜇框架提供的*spikingjelly.datasets.pad_sequence_collate*和*spikingjelly.datasets.padded_sequence_mask*可以很方便的对不等长数据进行对齐和还原。

示例代码：

```
import torch
from torch.utils.data import DataLoader
from spikingjelly.datasets import pad_sequence_collate, padded_sequence_mask, dvs128_
↪gesture
root='D:/datasets/DVS128Gesture'
train_set = dvs128_gesture.DVS128Gesture(root, data_type='frame', duration=1000000,␣
↪train=True)
for i in range(5):
    x, y = train_set[i]
    print(f'x[{i}].shape=[T, C, H, W]={x.shape}')
train_data_loader = DataLoader(train_set, collate_fn=pad_sequence_collate, batch_
↪size=5)
for x, y, x_len in train_data_loader:
    print(f'x.shape=[N, T, C, H, W]={tuple(x.shape)}')
    print(f'x_len={x_len}')
    mask = padded_sequence_mask(x_len)  # mask.shape = [T, N]
    print(f'mask=\n{mask.t().int()}')
    break
```

输出为：

```
The directory [D:/datasets/DVS128Gesture\duration_1000000] already exists.
x[0].shape=[T, C, H, W]=(6, 2, 128, 128)
x[1].shape=[T, C, H, W]=(6, 2, 128, 128)
x[2].shape=[T, C, H, W]=(5, 2, 128, 128)
x[3].shape=[T, C, H, W]=(5, 2, 128, 128)
x[4].shape=[T, C, H, W]=(7, 2, 128, 128)
x.shape=[N, T, C, H, W]=(5, 7, 2, 128, 128)
x_len=tensor([6, 6, 5, 5, 7])
mask=
tensor([[1, 1, 1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1, 1, 1]], dtype=torch.int32)
```

1.15.5 自定义积分方法

惊蜇框架支持用户自定义积分方法。用户只需要提供积分函数 `custom_integrate_function` 以及保存 frames 的文件夹名 `custom_integrated_frames_dir_name`。`custom_integrate_function` 是用户定义的函数，输入是 `events, H, W`，其中 `events` 是一个 pythono 字典，键为 `['t', 'x', 'y', 'p']` 值为 `numpy.ndarray` 类型。`H` 是数据高度，`W` 是数据宽度。例如，对于 DVS 手势数据集，H=128, W=128。这个函数的返回值应该是 frames。

`custom_integrated_frames_dir_name` 可以为 `None`，在这种情况下，保存 frames 的文件夹名会被设置成 `custom_integrate_function.__name__`。

例如，我们定义这样一种积分方式：随机将全部 events 一分为二，然后积分成 2 帧。我们可定义如下函数：

```
import spikingjelly.datasets as sjds
def integrate_events_to_2_frames_randomly(events: Dict, H: int, W: int):
    index_split = np.random.randint(low=0, high=events['t'].__len__())
    frames = np.zeros([2, 2, H, W])
    t, x, y, p = (events[key] for key in ('t', 'x', 'y', 'p'))
    frames[0] = sjds.integrate_events_segment_to_frame(x, y, p, H, W, 0, index_split)
    frames[1] = sjds.integrate_events_segment_to_frame(x, y, p, H, W, index_split,␣
↪events['t'].__len__())
    return frames
```

接下来创建数据集：

```
train_set = DVS128Gesture(root_dir, train=True, data_type='frame', custom_integrate_
↪function=integrate_events_to_2_frames_randomly)
```

运行完毕后，在 `root_dir` 目录下出现了 `integrate_events_to_2_frames_randomly` 文件夹，保存了我们的 frame 数据。

查看一下我们积分得到的数据：

```
from spikingjelly.datasets import play_frame
frame, label = train_set[500]
play_frame(frame)
```

惊蜇框架还支持其他的积分方式，阅读 API 文档以获取更多信息。

1.16 分类 DVS128 Gesture

本教程作者：fangwei123456

在上一个教程神经形态数据集处理 中，我们预处理了 DVS128 Gesture 数据集。接下来，我们将搭建 SNN 对 DVS128 Gesture 数据集进行分类，我们将使用 Incorporating Learnable Membrane Time Constant to Enhance Learning of Spiking Neural Networks[1] 一文中的网络，其中神经元使用 LIF 神经元，池化选用最大池化。

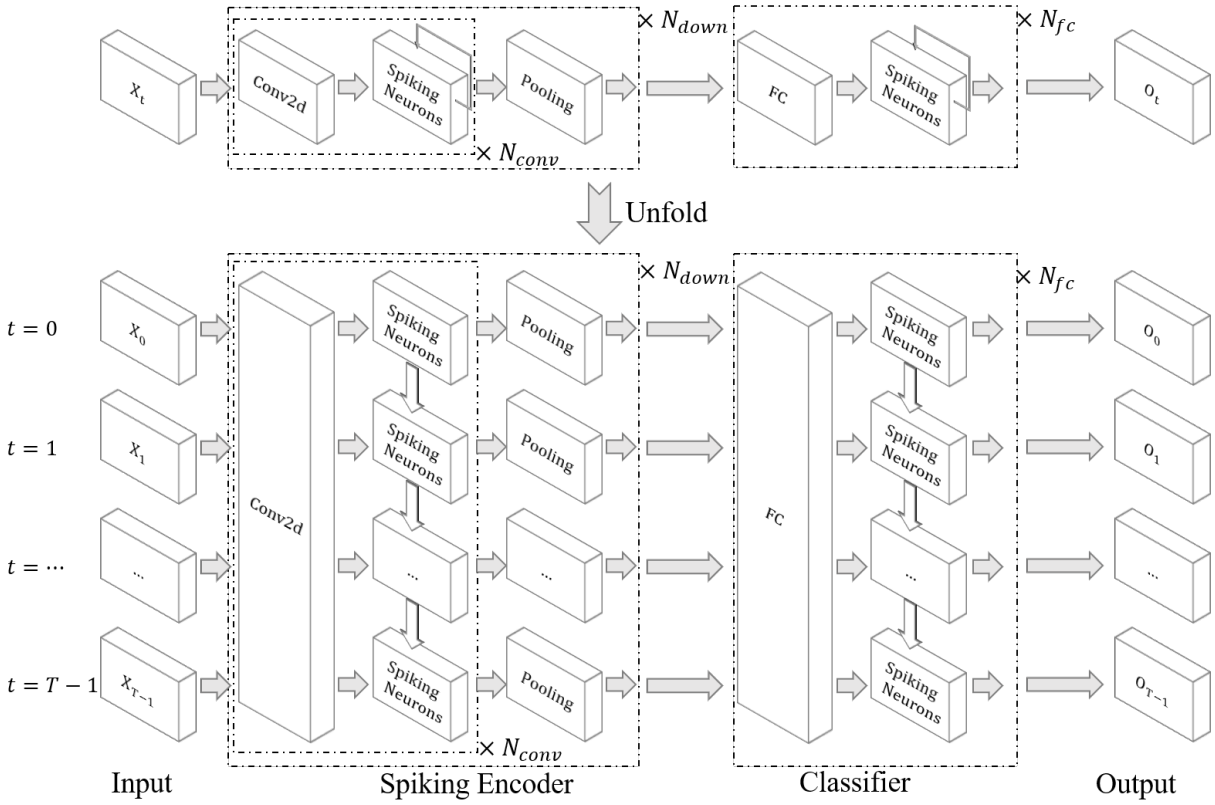
[1] Fang, Wei, et al. "Incorporating learnable membrane time constant to enhance learning of spiking neural networks." Proceedings of the IEEE/CVF

原文[Page 112, 1] 使用老版本的惊蜇框架，原始代码和训练日志可以在此处获取：Parametric-Leaky-Integrate-and-Fire-Spiking-Neuron

在本教程中，我们使用新版的惊蛰框架，将拥有更快的训练速度。

1.16.1 定义网路

原文[Page 112, 1] 使用下图所示的通用结构表示用于各个数据集的网络。



对于 DVS128 Gesture 数据集，$N_{conv} = 1, N_{down} = 5, N_{fc} = 2$。

具体的的网路结构为 *{c128k3s1-BN-LIF-MPk2s2}*5-DP-FC512-LIF-DP-FC110-LIF-APk10s10}*，其中 *APk10s10* 是额外增加的投票层。

符号的含义如下：

c128k3s1：`torch.nn.Conv2d(in_channels, out_channels=128, kernel_size=3, padding=1)`

BN：`torch.nn.BatchNorm2d(128)`

MPk2s2：`torch.nn.MaxPool2d(2, 2)`

DP：`spikingjelly.clock_driven.layer.Dropout(0.5)`

International Conference on Computer Vision. 2021.

FC512: `torch.nn.Linear(in_features, out_features=512`

APk10s10: `torch.nn.AvgPool1d(2, 2)`

简单起见，我们使用逐步仿真的方式定义网络，代码实现如下：

```
class VotingLayer(nn.Module):
    def __init__(self, voter_num: int):
        super().__init__()
        self.voting = nn.AvgPool1d(voter_num, voter_num)
    def forward(self, x: torch.Tensor):
        # x.shape = [N, voter_num * C]
        # ret.shape = [N, C]
        return self.voting(x.unsqueeze(1)).squeeze(1)

class PythonNet(nn.Module):
    def __init__(self, channels: int):
        super().__init__()
        conv = []
        conv.extend(PythonNet.conv3x3(2, channels))
        conv.append(nn.MaxPool2d(2, 2))
        for i in range(4):
            conv.extend(PythonNet.conv3x3(channels, channels))
            conv.append(nn.MaxPool2d(2, 2))
        self.conv = nn.Sequential(*conv)
        self.fc = nn.Sequential(
            nn.Flatten(),
            layer.Dropout(0.5),
            nn.Linear(channels * 4 * 4, channels * 2 * 2, bias=False),
            neuron.LIFNode(tau=2.0, surrogate_function=surrogate.ATan(), detach_
↪reset=True),
            layer.Dropout(0.5),
            nn.Linear(channels * 2 * 2, 110, bias=False),
            neuron.LIFNode(tau=2.0, surrogate_function=surrogate.ATan(), detach_
↪reset=True)
        )
        self.vote = VotingLayer(10)

    @staticmethod
    def conv3x3(in_channels: int, out_channels):
        return [
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1,␣
↪bias=False),
            nn.BatchNorm2d(out_channels),
            neuron.LIFNode(tau=2.0, surrogate_function=surrogate.ATan(), detach_
↪reset=True)
```

(续下页)

(接上页)

```
        ]
```

1.16.2 定义前向传播和损失

设置仿真时长为 `T`，batch size 为 `N`，则从 `DataLoader` 中获取的数据 `x.shape=[N, T, 2, 128, 128]`。我们定义的网络是按照逐步仿真的方式，最好先将 `x` 进行转换，转换为 `shape=[T, N, 2, 128, 128]`。

将 `x[t]` 送入网络，累加输出脉冲，除以总仿真时长，最终得到了脉冲发放频率 `out_spikes / x.shape[0]`，它是一个 `shape=[N, 11]` 的 tensor。

```
def forward(self, x: torch.Tensor):
    x = x.permute(1, 0, 2, 3, 4)  # [N, T, 2, H, W] -> [T, N, 2, H, W]
    out_spikes = self.vote(self.fc(self.conv(x[0])))
    for t in range(1, x.shape[0]):
        out_spikes += self.vote(self.fc(self.conv(x[t])))
    return out_spikes / x.shape[0]
```

损失定义为脉冲发放频率和 ont hot 形式标签的 MSE：

```
for frame, label in train_data_loader:
    optimizer.zero_grad()
    frame = frame.float().to(args.device)
    label = label.to(args.device)
    label_onehot = F.one_hot(label, 11).float()

    out_fr = net(frame)
    loss = F.mse_loss(out_fr, label_onehot)
    loss.backward()
    optimizer.step()

    functional.reset_net(net)
```

1.16.3 使用 CUDA 神经元和逐层传播

如果读者对惊蜇框架的传播模式不熟悉，建议先阅读之前的教程：*传播模式* 和*使用 CUDA 增强的神经元与逐层传播进行加速*。

逐步传播的代码通俗易懂，但速度较慢，现在让我们将原始网络改写为逐层传播：

```
import cupy

class CextNet(nn.Module):
```

(续下页)

(接上页)

```
    def __init__(self, channels: int):
        super().__init__()
        conv = []
        conv.extend(CextNet.conv3x3(2, channels))
        conv.append(layer.SeqToANNContainer(nn.MaxPool2d(2, 2)))
        for i in range(4):
            conv.extend(CextNet.conv3x3(channels, channels))
            conv.append(layer.SeqToANNContainer(nn.MaxPool2d(2, 2)))
        self.conv = nn.Sequential(*conv)
        self.fc = nn.Sequential(
            nn.Flatten(2),
            layer.MultiStepDropout(0.5),
            layer.SeqToANNContainer(nn.Linear(channels * 4 * 4, channels * 2 * 2,␣
↪bias=False)),
            neuron.MultiStepLIFNode(tau=2.0, surrogate_function=surrogate.ATan(),␣
↪detach_reset=True, backend='cupy'),
            layer.MultiStepDropout(0.5),
            layer.SeqToANNContainer(nn.Linear(channels * 2 * 2, 110, bias=False)),
            neuron.MultiStepLIFNode(tau=2.0, surrogate_function=surrogate.ATan(),␣
↪detach_reset=True, backend='cupy')
        )
        self.vote = VotingLayer(10)

    def forward(self, x: torch.Tensor):
        x = x.permute(1, 0, 2, 3, 4)  # [N, T, 2, H, W] -> [T, N, 2, H, W]
        out_spikes = self.fc(self.conv(x))  # shape = [T, N, 110]
        return self.vote(out_spikes.mean(0))

    @staticmethod
    def conv3x3(in_channels: int, out_channels):
        return [
            layer.SeqToANNContainer(
                nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1,␣
↪bias=False),
                nn.BatchNorm2d(out_channels),
            ),
            neuron.MultiStepLIFNode(tau=2.0, surrogate_function=surrogate.ATan(),␣
↪detach_reset=True, backend='cupy')
        ]
```

可以发现，网络的大致结构与逐步传播基本相同，所有的无状态的层，例如 `Conv2d`，都会被 `layer.SeqToANNContainer` 包装。前向传播的实现不需要时间上的循环：

```
def forward(self, x: torch.Tensor):
    x = x.permute(1, 0, 2, 3, 4)  # [N, T, 2, H, W] -> [T, N, 2, H, W]
    out_spikes = self.fc(self.conv(x))  # shape = [T, N, 110]
    return self.vote(out_spikes.mean(0))
```

1.16.4 代码细节

为了便于调试，让我们在代码中加入大量的超参数：

```
parser = argparse.ArgumentParser(description='Classify DVS128 Gesture')
parser.add_argument('-T', default=16, type=int, help='simulating time-steps')
parser.add_argument('-device', default='cuda:0', help='device')
parser.add_argument('-b', default=16, type=int, help='batch size')
parser.add_argument('-epochs', default=64, type=int, metavar='N',
                    help='number of total epochs to run')
parser.add_argument('-j', default=4, type=int, metavar='N',
                    help='number of data loading workers (default: 4)')
parser.add_argument('-channels', default=128, type=int, help='channels of Conv2d in␣
↪SNN')
parser.add_argument('-data_dir', type=str, help='root dir of DVS128 Gesture dataset')
parser.add_argument('-out_dir', type=str, help='root dir for saving logs and␣
↪checkpoint')

parser.add_argument('-resume', type=str, help='resume from the checkpoint path')
parser.add_argument('-amp', action='store_true', help='automatic mixed precision␣
↪training')
parser.add_argument('-cupy', action='store_true', help='use CUDA neuron and multi-
↪step forward mode')


parser.add_argument('-opt', type=str, help='use which optimizer. SDG or Adam')
parser.add_argument('-lr', default=0.001, type=float, help='learning rate')
parser.add_argument('-momentum', default=0.9, type=float, help='momentum for SGD')
parser.add_argument('-lr_scheduler', default='CosALR', type=str, help='use which␣
↪schedule. StepLR or CosALR')
parser.add_argument('-step_size', default=32, type=float, help='step_size for StepLR')
parser.add_argument('-gamma', default=0.1, type=float, help='gamma for StepLR')
parser.add_argument('-T_max', default=32, type=int, help='T_max for CosineAnnealingLR
↪')
```

使用混合精度训练，可以大幅度提升速度，减少显存消耗：

```
if args.amp:
```

(续下页)

(接上页)

```
    with amp.autocast():
        out_fr = net(frame)
        loss = F.mse_loss(out_fr, label_onehot)
    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()
else:
    out_fr = net(frame)
    loss = F.mse_loss(out_fr, label_onehot)
    loss.backward()
    optimizer.step()
```

我们的网络将支持断点续训：

```
#...........
if args.resume:
    checkpoint = torch.load(args.resume, map_location='cpu')
    net.load_state_dict(checkpoint['net'])
    optimizer.load_state_dict(checkpoint['optimizer'])
    lr_scheduler.load_state_dict(checkpoint['lr_scheduler'])
    start_epoch = checkpoint['epoch'] + 1
    max_test_acc = checkpoint['max_test_acc']
# ...

for epoch in range(start_epoch, args.epochs):
# train...

# test...

    checkpoint = {
        'net': net.state_dict(),
        'optimizer': optimizer.state_dict(),
        'lr_scheduler': lr_scheduler.state_dict(),
        'epoch': epoch,
        'max_test_acc': max_test_acc
    }

    # ...

    torch.save(checkpoint, os.path.join(out_dir, 'checkpoint_latest.pth'))
```

1.16.5 运行训练

完整的代码位于 spikingjelly.clock_driven.examples.classify_dvsg。

我们在 'Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz' 的 CPU 和 *GeForce RTX 2080 Ti* 的 GPU 上运行网络。我们使用的超参数几乎与原文[Page 112, 1] 一致，但略有区别：我们使用 `T=16` 而原文[Page 112, 1] 使用 `T=20`，因为 *GeForce RTX 2080 Ti* 的 12G 显存不够使用 `T=20`；此外，我们还开启了自动混合精度训练，正确率可能会略微低于全精度训练。

运行一下逐步模式的网络：

```
(test-env) root@de41f92009cf3011eb0ac59057a81652d2d0-fangw1714-0:/userhome/test#␣
↪python -m spikingjelly.clock_driven.examples.classify_dvsg -data_dir /userhome/
↪datasets/DVS128Gesture -out_dir ./logs -amp -opt Adam -device cuda:0 -lr_scheduler␣
↪CosALR -T_max 64 -epochs 256
Namespace(T=16, T_max=64, amp=True, b=16, cupy=False, channels=128, data_dir='/
↪userhome/datasets/DVS128Gesture', device='cuda:0', epochs=256, gamma=0.1, j=4, lr=0.
↪001, lr_scheduler='CosALR', momentum=0.9, opt='Adam', out_dir='./logs', resume=None,
↪ step_size=32)
PythonNet(
  (conv): Sequential(
    (0): Conv2d(2, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
    (2): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
    (6): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
    (9): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
    (10): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
```

(续下页)

(接上页)

```
    )
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (12): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
    (13): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
    (14): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (16): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
    (17): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
    (18): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (19): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Dropout(p=0.5)
    (2): Linear(in_features=2048, out_features=512, bias=False)
    (3): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (4): Dropout(p=0.5)
    (5): Linear(in_features=512, out_features=110, bias=False)
    (6): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
  )
  (vote): VotingLayer(
    (voting): AvgPool1d(kernel_size=(10,), stride=(10,), padding=(0,))
  )
)
The directory [/userhome/datasets/DVS128Gesture/frames_number_16_split_by_number]␣
↪already exists.
The directory [/userhome/datasets/DVS128Gesture/frames_number_16_split_by_number]␣
```

(续下页)

(接上页)

```
↪already exists.
Mkdir ./logs/T_16_b_16_c_128_Adam_lr_0.001_CosALR_64_amp.
Namespace(T=16, T_max=64, amp=True, b=16, cupy=False, channels=128, data_dir='/
↪userhome/datasets/DVS128Gesture', device='cuda:0', epochs=256, gamma=0.1, j=4, lr=0.
↪001, lr_scheduler='CosALR', momentum=0.9, opt='Adam', out_dir='./logs', resume=None,
↪ step_size=32)
epoch=0, train_loss=0.06680945929599134, train_acc=0.4032534246575342, test_loss=0.
↪04891310722774102, test_acc=0.6180555555555556, max_test_acc=0.6180555555555556,␣
↪total_time=27.759592294692993
```

可以发现，一个 epoch 用时为 27.76s。中断训练，让我们换成速度更快的模式：

```
(test-env) root@de41f92009cf3011eb0ac59057a81652d2d0-fangw1714-0:/userhome/test#␣
↪python -m spikingjelly.clock_driven.examples.classify_dvsg -data_dir /userhome/
↪datasets/DVS128Gesture -out_dir ./logs -amp -opt Adam -device cuda:0 -lr_scheduler␣
↪CosALR -T_max 64 -cupy -epochs 256
Namespace(T=16, T_max=64, amp=True, b=16, cupy=True, channels=128, data_dir='/
↪userhome/datasets/DVS128Gesture', device='cuda:0', epochs=256, gamma=0.1, j=4, lr=0.
↪001, lr_scheduler='CosALR', momentum=0.9, opt='Adam', out_dir='./logs', resume=None,
↪ step_size=32)
CextNet(
  (conv): Sequential(
    (0): SeqToANNContainer(
      (module): Sequential(
        (0): Conv2d(2, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
↪function=ATan, alpha=2.0 tau=2.0)
    (2): SeqToANNContainer(
      (module): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
    )
    (3): SeqToANNContainer(
      (module): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
```

(续下页)

(接上页)

```
    (4): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
↪function=ATan, alpha=2.0 tau=2.0)
    (5): SeqToANNContainer(
      (module): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
    )
    (6): SeqToANNContainer(
      (module): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (7): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
↪function=ATan, alpha=2.0 tau=2.0)
    (8): SeqToANNContainer(
      (module): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
    )
    (9): SeqToANNContainer(
      (module): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (10): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
↪function=ATan, alpha=2.0 tau=2.0)
    (11): SeqToANNContainer(
      (module): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
    )
    (12): SeqToANNContainer(
      (module): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (13): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
```

(续下页)

(接上页)

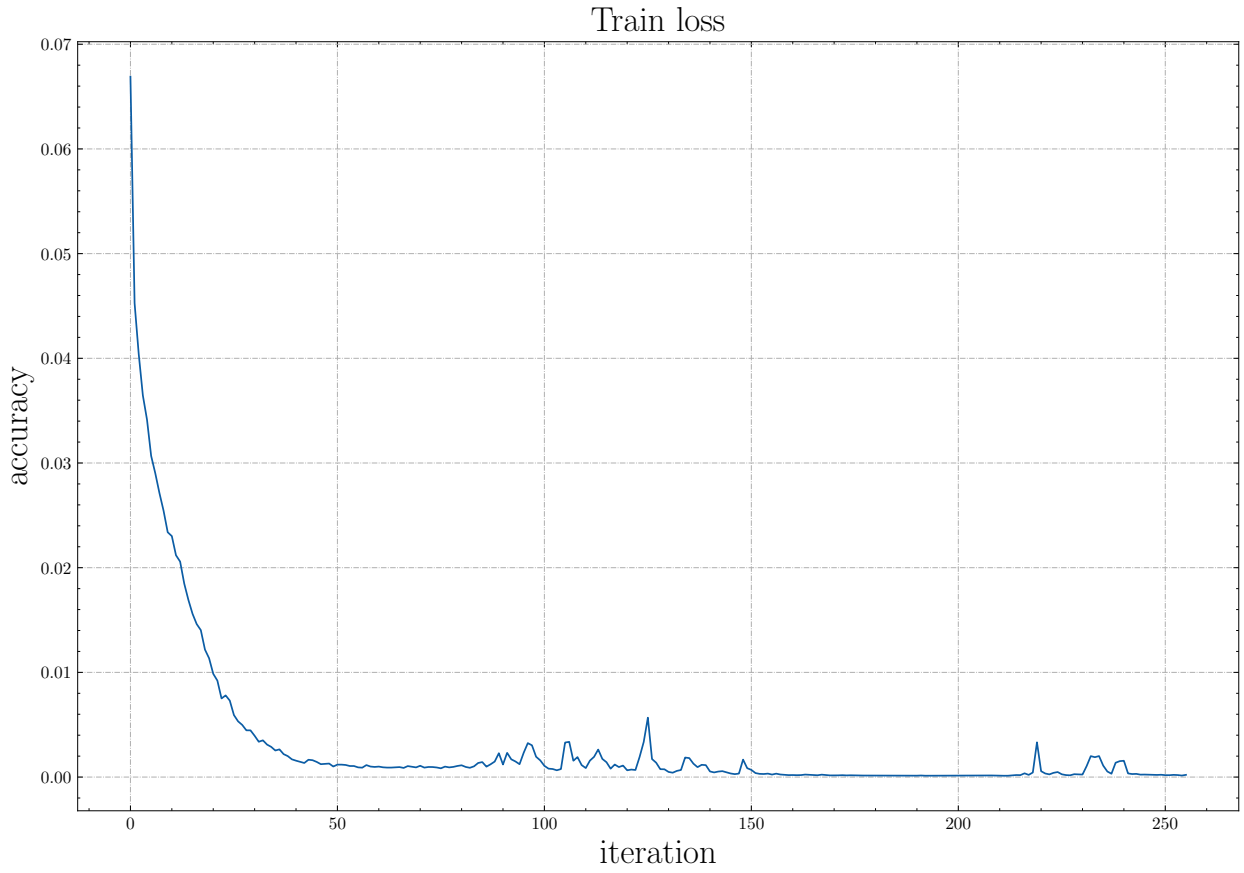
```
↪function=ATan, alpha=2.0 tau=2.0)
    (14): SeqToANNContainer(
      (module): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
    )
  )
  (fc): Sequential(
    (0): Flatten(start_dim=2, end_dim=-1)
    (1): MultiStepDropout(p=0.5)
    (2): SeqToANNContainer(
      (module): Linear(in_features=2048, out_features=512, bias=False)
    )
    (3): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
↪function=ATan, alpha=2.0 tau=2.0)
    (4): MultiStepDropout(p=0.5)
    (5): SeqToANNContainer(
      (module): Linear(in_features=512, out_features=110, bias=False)
    )
    (6): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
↪function=ATan, alpha=2.0 tau=2.0)
  )
  (vote): VotingLayer(
    (voting): AvgPool1d(kernel_size=(10,), stride=(10,), padding=(0,))
  )
)
The directory [/userhome/datasets/DVS128Gesture/frames_number_16_split_by_number]␣
↪already exists.
The directory [/userhome/datasets/DVS128Gesture/frames_number_16_split_by_number]␣
↪already exists.
Mkdir ./logs/T_16_b_16_c_128_Adam_lr_0.001_CosALR_64_amp_cupy.
Namespace(T=16, T_max=64, amp=True, b=16, cupy=True, channels=128, data_dir='/
↪userhome/datasets/DVS128Gesture', device='cuda:0', epochs=256, gamma=0.1, j=4, lr=0.
↪001, lr_scheduler='CosALR', momentum=0.9, opt='Adam', out_dir='./logs', resume=None,
↪ step_size=32)
epoch=0, train_loss=0.06690179117738385, train_acc=0.4092465753424658, test_loss=0.
↪049108295158172645, test_acc=0.6145833333333334, max_test_acc=0.6145833333333334,␣
↪total_time=18.169376373291016

...

Namespace(T=16, T_max=64, amp=True, b=16, cupy=True, channels=128, data_dir='/
↪userhome/datasets/DVS128Gesture', device='cuda:0', epochs=256, gamma=0.1, j=4, lr=0.
↪001, lr_scheduler='CosALR', momentum=0.9, opt='Adam', out_dir='./logs', resume=None,
```

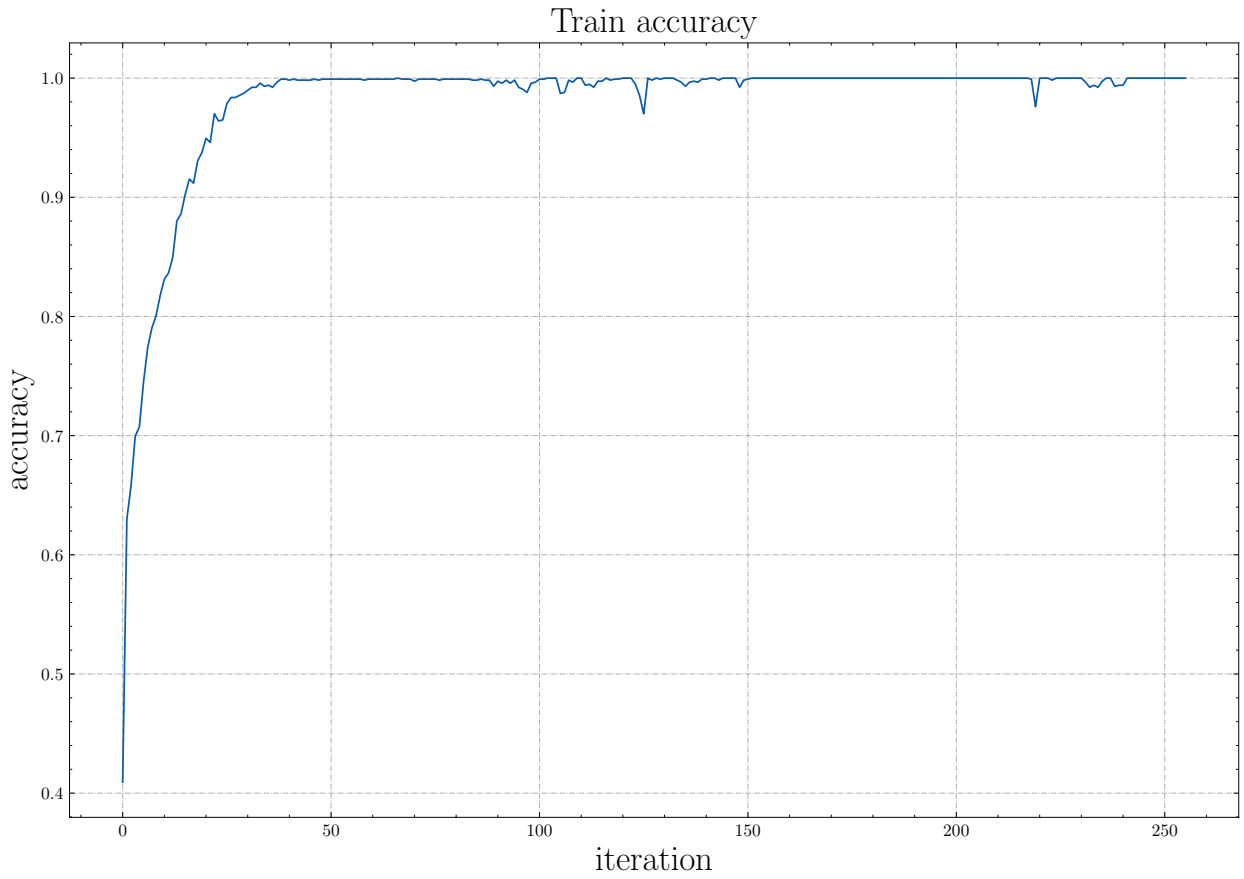
(续下页)

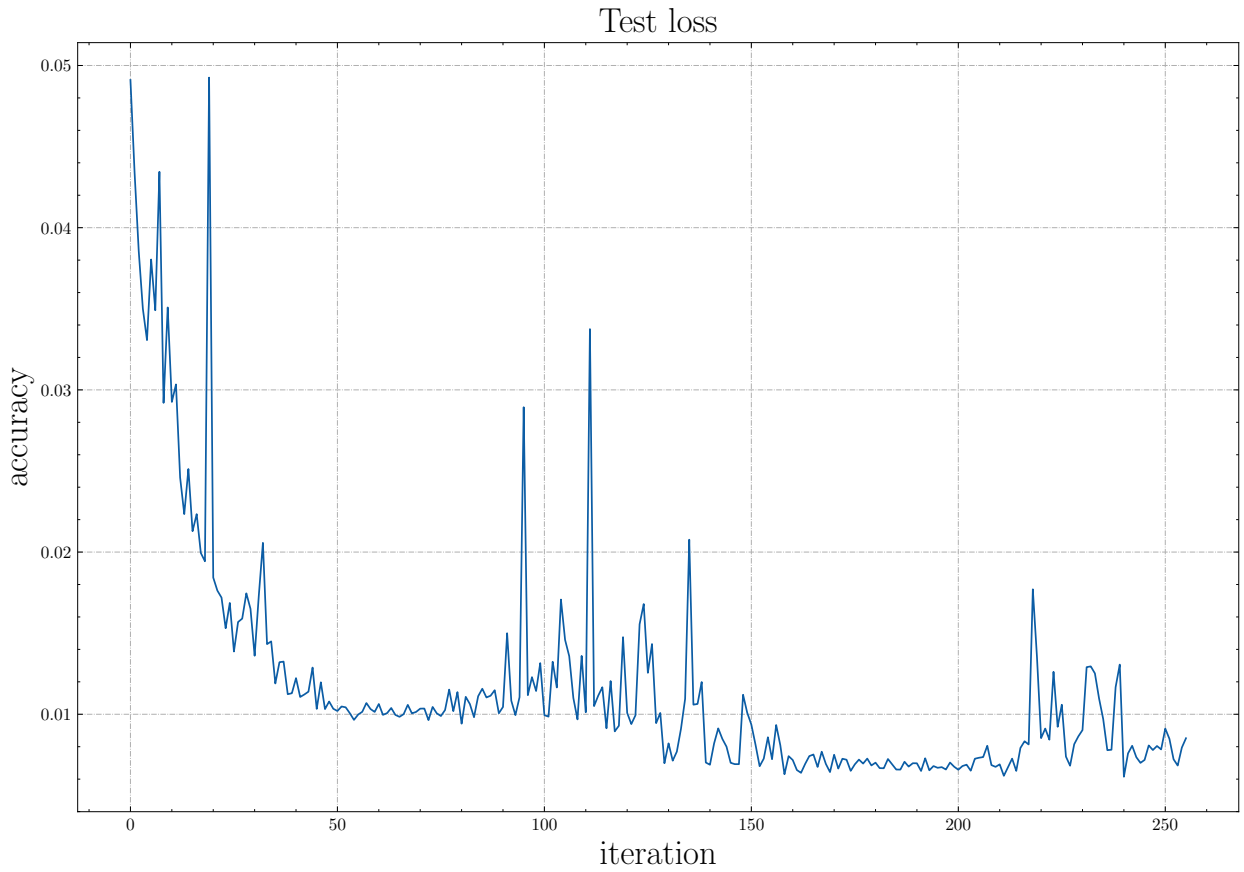
(接上页)

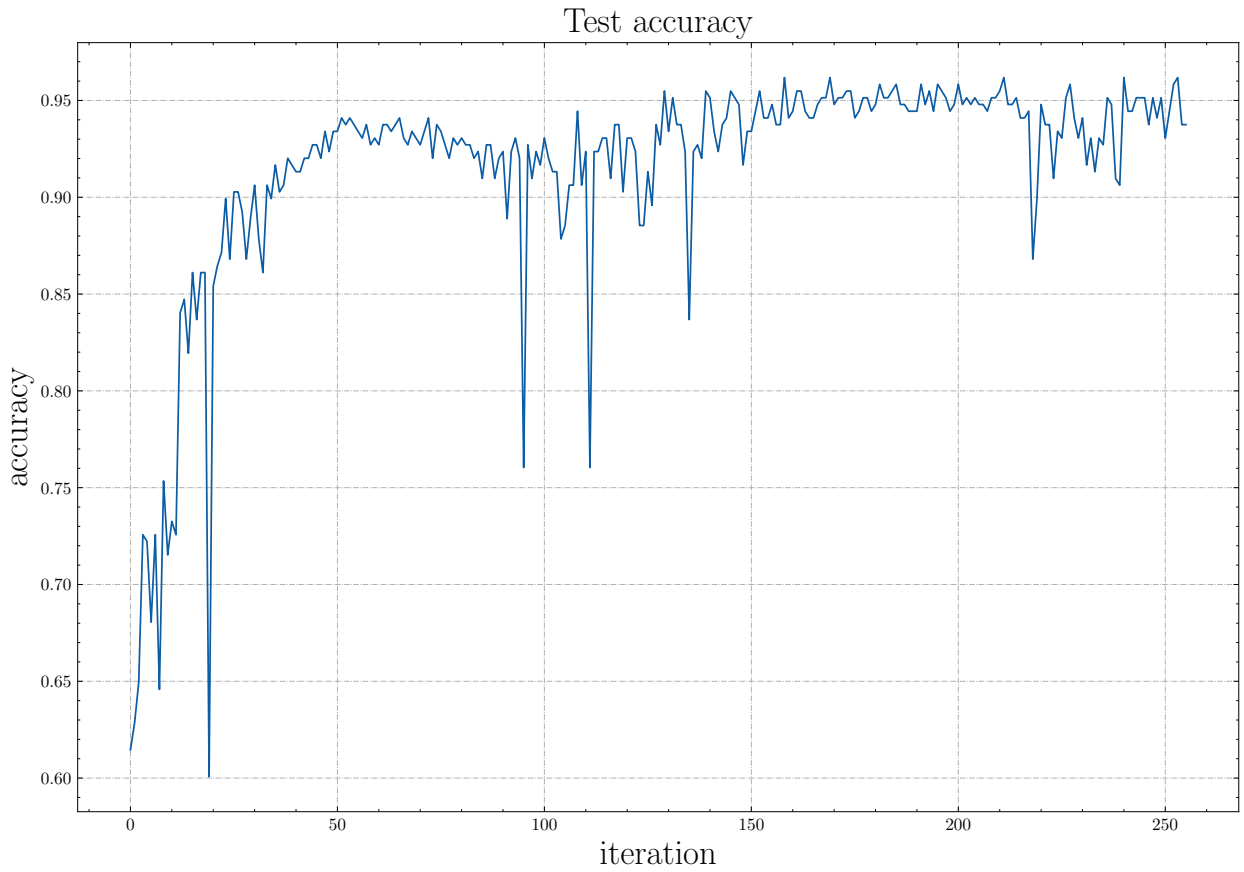
```
↪ step_size=32)
epoch=255, train_loss=0.00021228195577325645, train_acc=1.0, test_loss=0.
↪00852209396485576, test_acc=0.9375, max_test_acc=0.9618055555555556, total_time=17.
↪49005389213562
```

训练一个 epoch 耗时为 18.17s，比逐步传播的 27.76s 快了约 10s。训练 256 个 epoch，我们可以达到最高 96.18% 的正确率，得到的训练曲线如下：




Train accuracy
accuracy
iteration
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
50
100
150
200
250


Test loss
accuracy
iteration
0.05
0.04
0.03
0.02
0.01
0
50
100
150
200
250



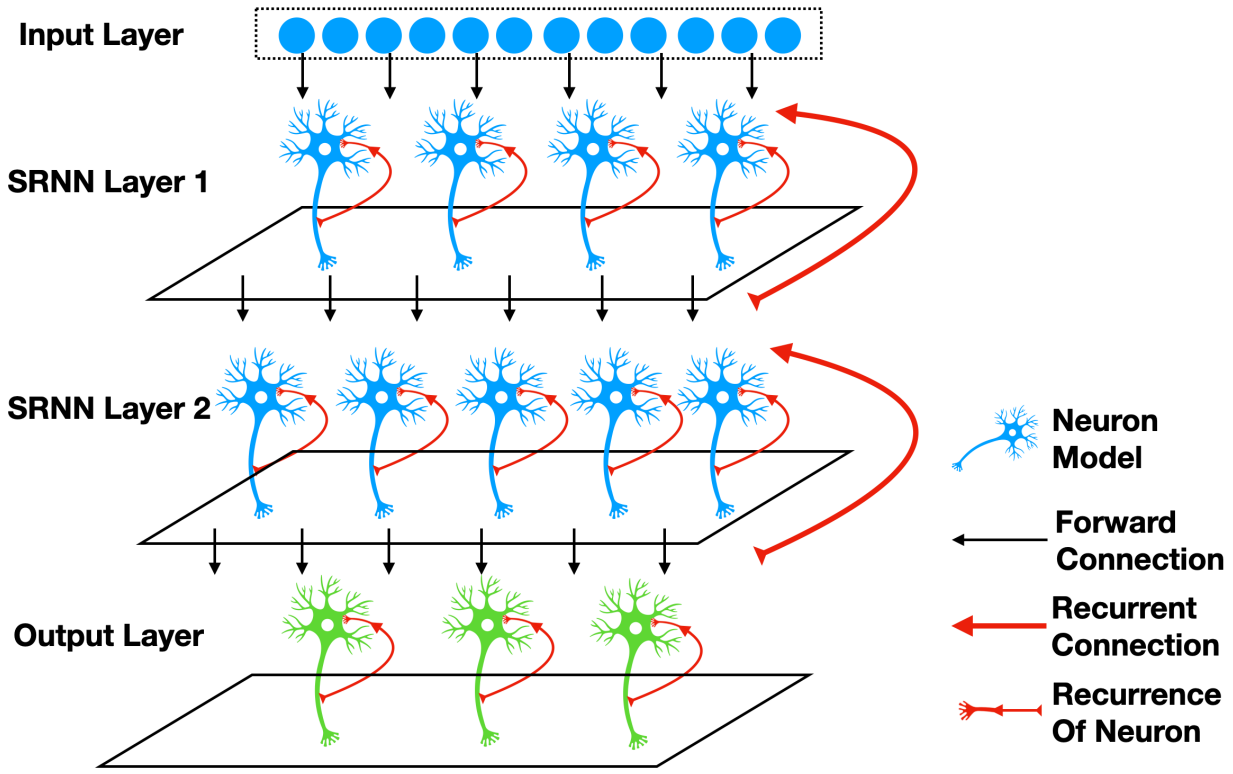
1.17 自连接和有状态突触

本教程作者：fangwei123456

1.17.1 自连接模块

自连接指的是从输出到输入的连接，例如[1] 一文中的 SRNN(recurrent networks of spiking neurons)，如下图所示：

[1] 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.



使用惊蜇框架很容易构建出带有自连接的模块。考虑最简单的一种情况，我们给神经元增加一个回路，使得它在 t 时刻的输出 $s[t]$，会与下一个时刻的外界输入 $x[t+1]$ 相加，共同作为输入。这可以由*spikingjelly.clock_driven.layer.ElementWiseRecurrentContainer*轻松实现。`ElementWiseRecurrentContainer` 是一个包装器，给任意的 `sub_module` 增加一个额外的自连接。连接的形式可以使用用户自定义的逐元素函数操作 $z = f(x, y)$ 来实现。记 $x[t]$ 为 t 时刻整个模块的输入，$i[t]$ 和 $y[t]$ 是 `sub_module` 的输入和输出（注意 $y[t]$ 同时也是整个模块的输出），则

$$i[t] = f(x[t], y[t-1])$$

其中 f 是用户自定义的逐元素操作。默认 $y[-1] = 0$。

现在让我们用 `ElementWiseRecurrentContainer` 来包装一个 IF 神经元，逐元素操作设置为加法，因而

$$i[t] = x[t] + y[t-1].$$

我们使用软重置，且给与 $x[t] = [1.5, 0, ..., 0]$ 的输入：

```
T = 8
def element_wise_add(x, y):
    return x + y
net = ElementWiseRecurrentContainer(neuron.IFNode(v_reset=None), element_wise_add)
print(net)
x = torch.zeros([T])
x[0] = 1.5
```

(续下页)

(接上页)

```
for t in range(T):
    print(t, f'x[t]={x[t]}, s[t]={net(x[t])}')

functional.reset_net(net)
```

输出为：

```
ElementWiseRecurrentContainer(
  element-wise function=<function element_wise_add at 0x000001FE0F7968B0>
  (sub_module): IFNode(
    v_threshold=1.0, v_reset=None, detach_reset=False
    (surrogate_function): Sigmoid(alpha=1.0, spiking=True)
  )
)
0 x[t]=1.5, s[t]=1.0
1 x[t]=0.0, s[t]=1.0
2 x[t]=0.0, s[t]=1.0
3 x[t]=0.0, s[t]=1.0
4 x[t]=0.0, s[t]=1.0
5 x[t]=0.0, s[t]=1.0
6 x[t]=0.0, s[t]=1.0
7 x[t]=0.0, s[t]=1.0
```

可以发现，由于存在自连接，即便 $t \geq 1$ 时 $x[t] = 0$，由于输出的脉冲能传回到输入，神经元也能持续释放脉冲。

可以使用*spikingjelly.clock_driven.layer.LinearRecurrentContainer*实现更复杂的全连接形式的自连接。

1.17.2 有状态的突触

[2] [3] 等文章使用有状态的突触。将*spikingjelly.clock_driven.layer.SynapseFilter*放在普通无状态突触的后面，对突触输出的电流进行滤波，就可以得到有状态的突触，例如：

```
stateful_conv = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=3, padding=1, stride=1),
    SynapseFilter(tau=100, learnable=True)
)
```

[2] Diehl P U, Cook M. Unsupervised learning of digit recognition using spike-timing-dependent plasticity[J]. Frontiers in computational neuroscience, 2015, 9: 99.

[3] Fang H, Shrestha A, Zhao Z, et al. Exploiting Neuron and Synapse Filter Dynamics in Spatial Temporal Learning of Deep Spiking Neural Network[J].

1.17.3 Sequential FashionMNIST 上的对比实验

接下来让我们在 Sequential FashionMNIST 上做一个简单的实验，验证自连接和有状态突触是否有助于改善网络的记忆能力。Sequential FashionMNIST 指的是将原始的 FashionMNIST 图片一行一行或者一列一列，而不是整个图片，作为输入。在这种情况下，网络必须具有一定的记忆能力，才能做出正确的分类。我们将会把图片一列一列的输入，这样对网络而言，就像是从左到右“阅读”一样，如下图所示：

下图中展示了被读入的列：

首先导入相关的包：

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.datasets
from spikingjelly.clock_driven.model import train_classify
from spikingjelly.clock_driven import neuron, surrogate, layer
from spikingjelly.clock_driven.functional import seq_to_ann_forward
from torchvision import transforms
import os, argparse

try:
    import cupy
    backend = 'cupy'
except ImportError:
    backend = 'torch'
```

我们定义一个普通的前馈网络 Net：

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28, 32)
        self.sn1 = neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), detach_
↪reset=True, backend=backend)
        self.fc2 = nn.Linear(32, 10)
        self.sn2 = neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), detach_
↪reset=True, backend=backend)

    def forward(self, x: torch.Tensor):
        # x.shape = [N, C, H, W]
        x.squeeze_(1)  # [N, H, W]
        x = x.permute(2, 0, 1)  # [W, N, H]
        x = seq_to_ann_forward(x, self.fc1)
        x = self.sn1(x)
        x = seq_to_ann_forward(x, self.fc2)
        x = self.sn2(x)
        return x.mean(0)
```

我们在 Net 的第一层脉冲神经元后增加一个*spikingjelly.clock_driven.layer.SynapseFilter*，得到一个新的网络 StatefulSynapseNet：

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

(续下页)

(接上页)

```
        self.fc1 = nn.Linear(28, 32)
        self.sn1 = neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), detach_
↪reset=True, backend=backend)
        self.sy1 = layer.MultiStepContainer(layer.SynapseFilter(tau=2.,␣
↪learnable=True))
        self.fc2 = nn.Linear(32, 10)
        self.sn2 = neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), detach_
↪reset=True, backend=backend)

    def forward(self, x: torch.Tensor):
        # x.shape = [N, C, H, W]
        x.squeeze_(1)  # [N, H, W]
        x = x.permute(2, 0, 1)  # [W, N, H]
        x = self.fc1(x)
        x = self.sn1(x)
        x = self.sy1(x)
        x = self.fc2(x)
        x = self.sn2(x)
        return x.mean(0)
```

我们给 Net 的第一层脉冲神经元增加一个反馈连接*spikingjelly.clock_driven.layer.LinearRecurrentContainer*得到 FeedBackNet：

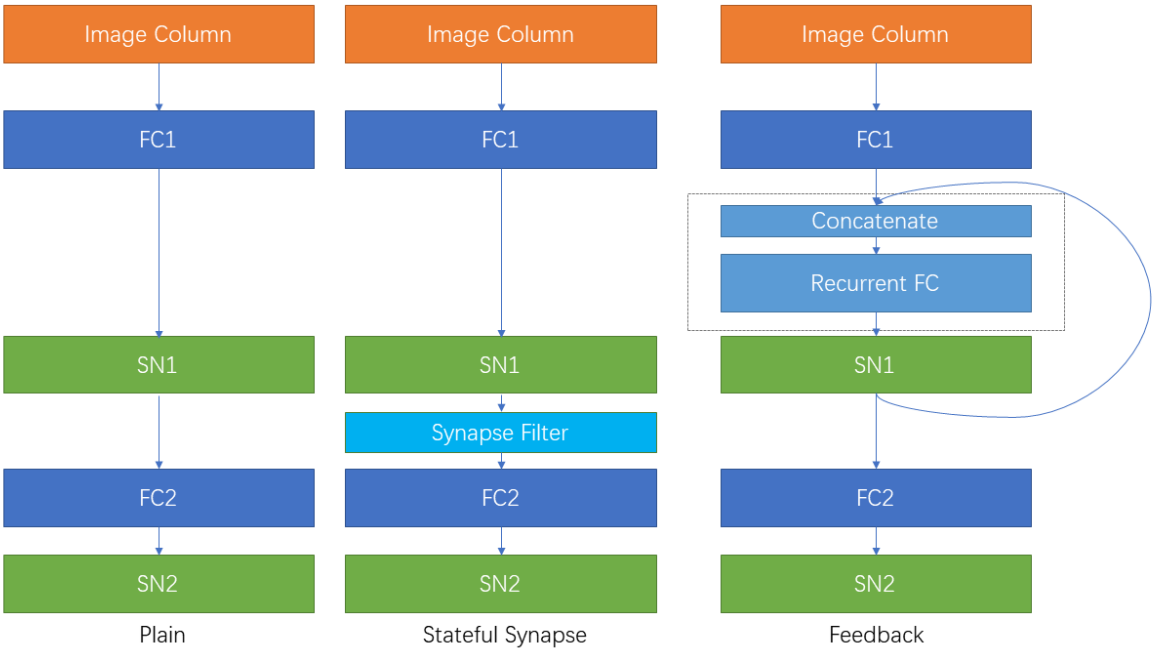
```
class FeedBackNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28, 32)
        self.sn1 = layer.MultiStepContainer(
            layer.LinearRecurrentContainer(
                neuron.IFNode(surrogate_function=surrogate.ATan(), detach_reset=True),
                32, 32
            )
        )
        self.fc2 = nn.Linear(32, 10)
        self.sn2 = neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), detach_
↪reset=True, backend=backend)

    def forward(self, x: torch.Tensor):
        # x.shape = [N, C, H, W]
        x.squeeze_(1)  # [N, H, W]
        x = x.permute(2, 0, 1)  # [W, N, H]
        x = seq_to_ann_forward(x, self.fc1)
        x = self.sn1(x)
```

(续下页)

(接上页)

```
        x = seq_to_ann_forward(x, self.fc2)
        x = self.sn2(x)
        return x.mean(0)
```

下图展示了 3 种网络的结构：



完整的代码位于 spikingjelly.clock_driven.examples.rsnn_sequential_fmnist。我们可以通过命令行直接运行。运行参数为：

```
(pytorch-env) PS C:/Users/fw> python -m spikingjelly.clock_driven.examples.rsnn_
↪sequential_fmnist --h
usage: rsnn_sequential_fmnist.py [-h] [--data-path DATA_PATH] [--device DEVICE] [-b␣
↪BATCH_SIZE] [--epochs N] [-j N]
                                 [--lr LR] [--opt OPT] [--lrs LRS] [--step-size STEP_
↪SIZE] [--step-gamma STEP_GAMMA]
                                 [--cosa-tmax COSA_TMAX] [--momentum M] [--wd W] [--
↪output-dir OUTPUT_DIR]
                                 [--resume RESUME] [--start-epoch N] [--cache-
↪dataset] [--amp] [--tb] [--model MODEL]

PyTorch Classification Training

optional arguments:
  -h, --help            show this help message and exit
  --data-path DATA_PATH
```

(续下页)

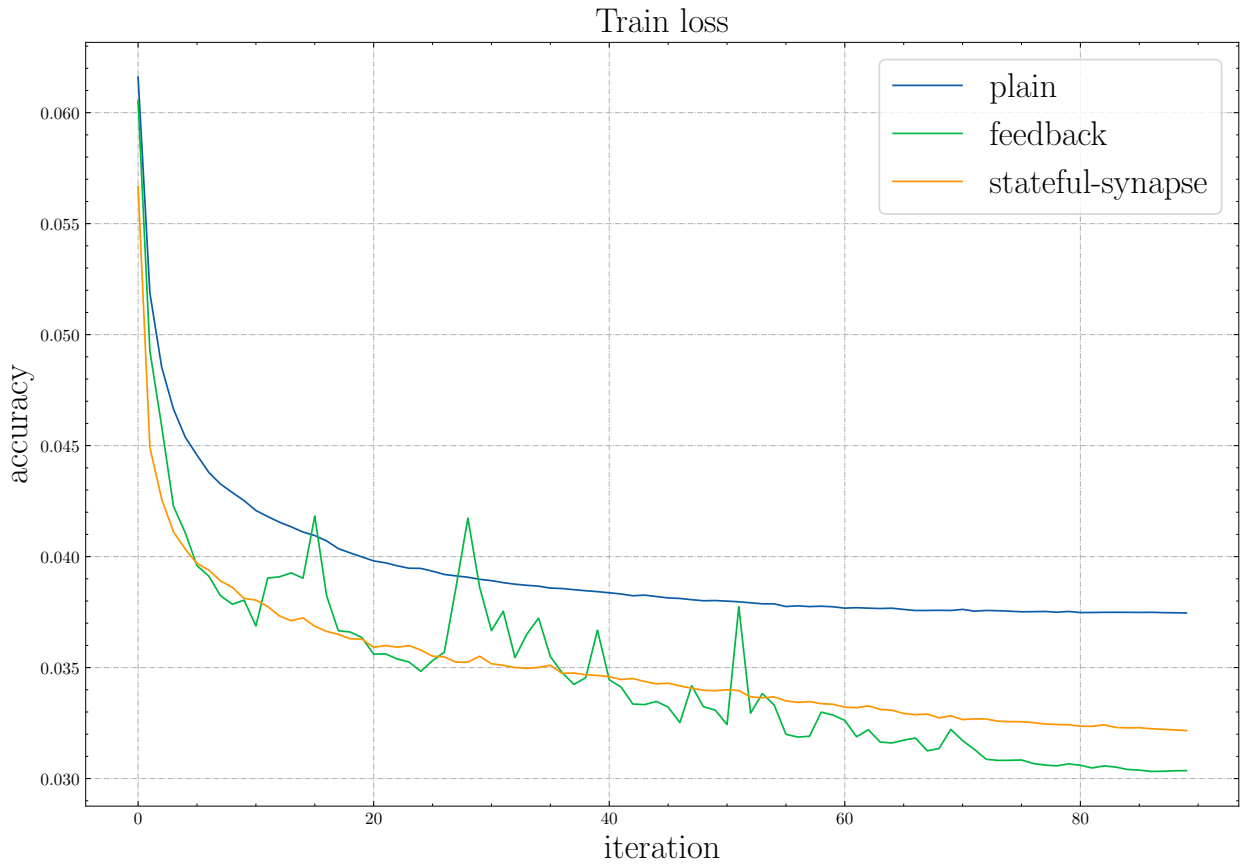
(接上页)

```
                        dataset
  --device DEVICE       device
  -b BATCH_SIZE, --batch-size BATCH_SIZE
  --epochs N            number of total epochs to run
  -j N, --workers N     number of data loading workers (default: 16)
  --lr LR               initial learning rate
  --opt OPT             optimizer (sgd or adam)
  --lrs LRS             lr schedule (cosa(CosineAnnealingLR), step(StepLR)) or None
  --step-size STEP_SIZE
                        step_size for StepLR
  --step-gamma STEP_GAMMA
                        gamma for StepLR
  --cosa-tmax COSA_TMAX
                        T_max for CosineAnnealingLR. If none, it will be set to epochs
  --momentum M          Momentum for SGD
  --wd W, --weight-decay W
                        weight decay (default: 0)
  --output-dir OUTPUT_DIR
                        path where to save
  --resume RESUME       resume from checkpoint
  --start-epoch N       start epoch
  --cache-dataset       Cache the datasets for quicker initialization. It also␣
↪serializes the transforms
  --amp                 Use AMP training
  --tb                  Use TensorBoard to record logs
  --model MODEL         "plain", "feedback", or "stateful-synapse"
```

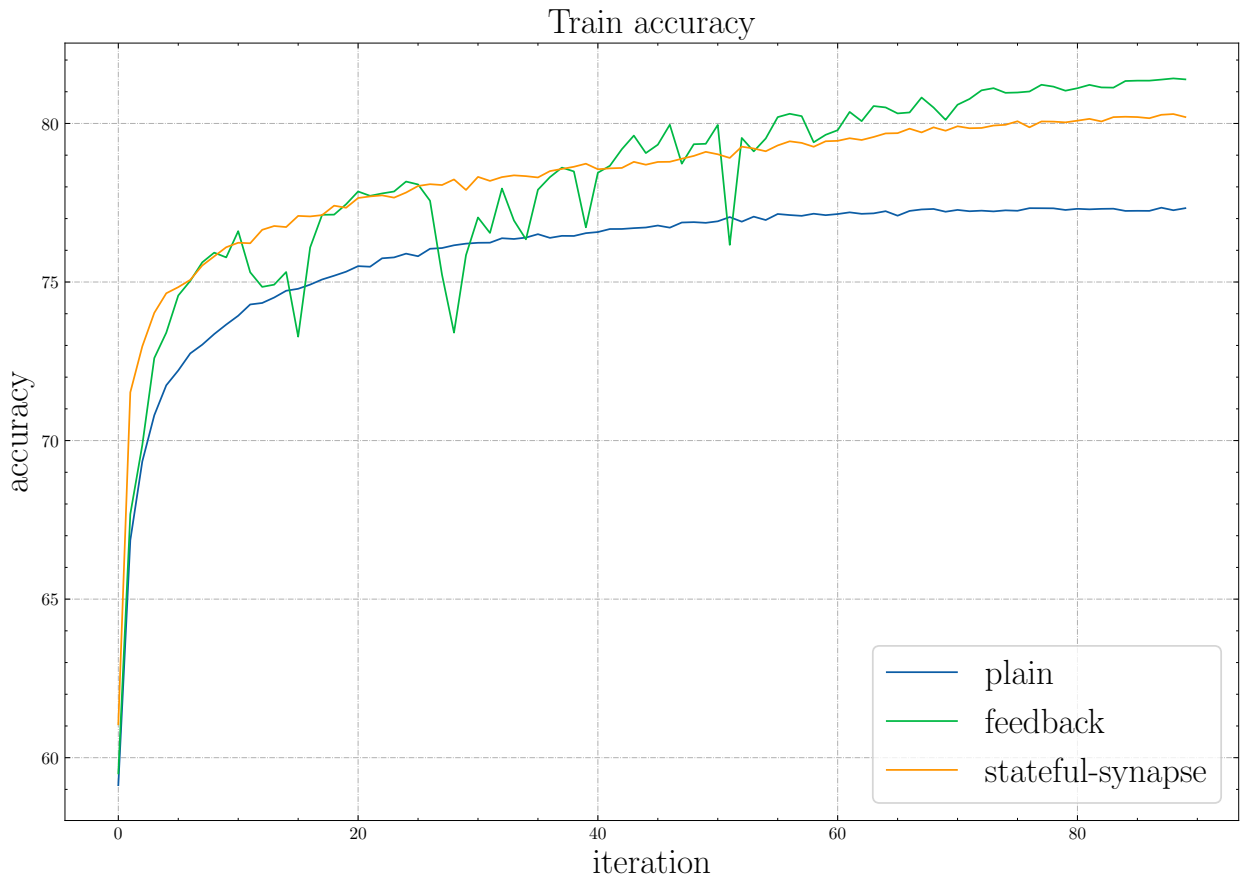
分别训练 3 个模型：

```
python -m spikingjelly.clock_driven.examples.rsnn_sequential_fmnist --data-path /raid/
↪wfang/datasets/FashionMNIST --tb --device cuda:0 --amp --model plain

python -m spikingjelly.clock_driven.examples.rsnn_sequential_fmnist --data-path /raid/
↪wfang/datasets/FashionMNIST --tb --device cuda:1 --amp --model feedback

python -m spikingjelly.clock_driven.examples.rsnn_sequential_fmnist --data-path /raid/
↪wfang/datasets/FashionMNIST --tb --device cuda:2 --amp --model stateful-synapse
```

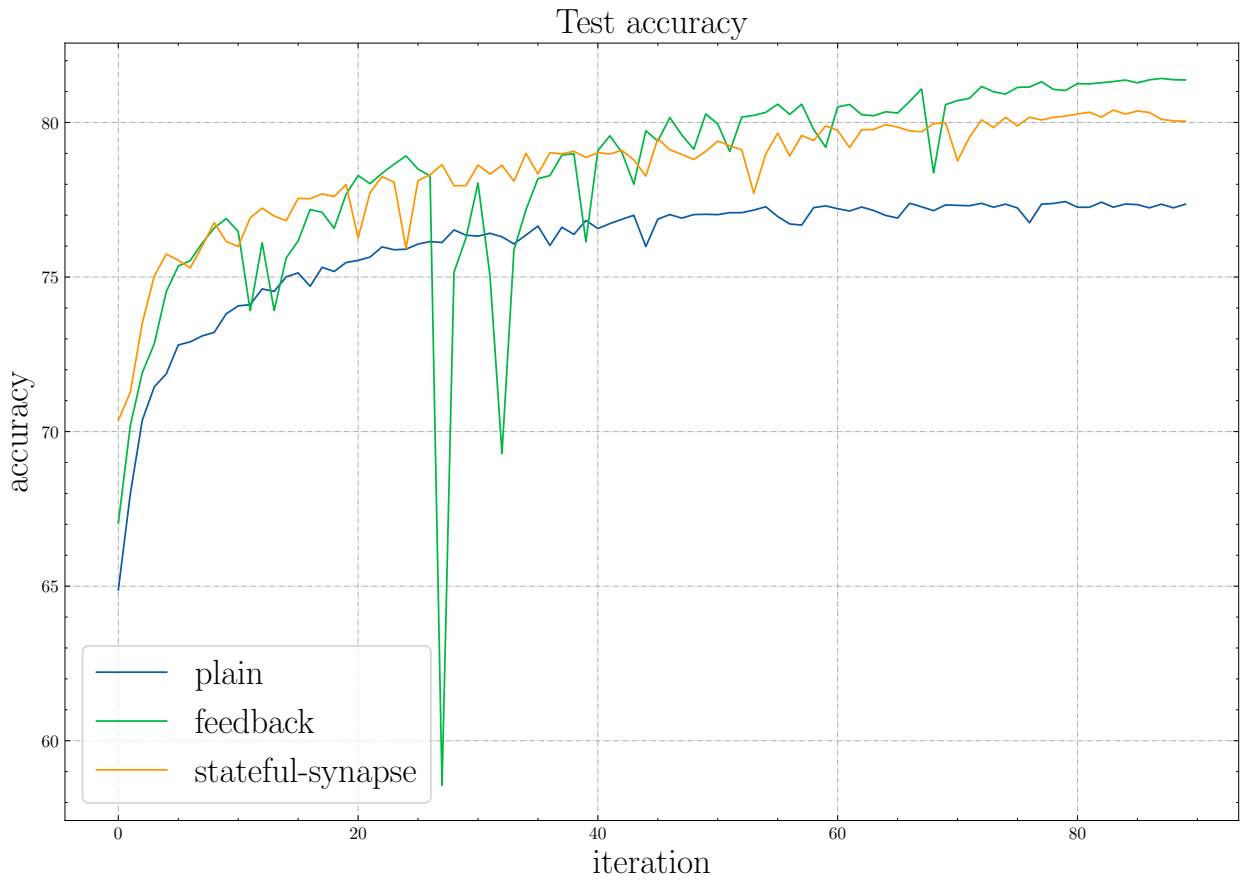
训练集损失为：


Train loss
plain
feedback
stateful-synapse
accuracy
iteration
0.060
0.055
0.050
0.045
0.040
0.035
0.030
0
20
40
60
80

训练集正确率为:


Train accuracy
accuracy
iteration
plain
feedback
stateful-synapse
80
75
70
65
60
0
20
40
60
80

测试集正确率为:



可以发现，`feedback` 和 `stateful-synapse` 的性能都高于 `plain`，表明自连接和有状态突触都有助于提升网络的记忆能力。

1.18 训练大规模 SNN

本教程作者：fangwei123456

1.18.1 使用 spikingjelly.clock_driven.model

在*spikingjelly.clock_driven.model* 中定义了一些常用的网络，下面以*spikingjelly.clock_driven.model.spiking_resnet* 为例展示如何使用这些网络。

大多数*spikingjelly.clock_driven.model* 中的网络，都提供单步和多步两种网络。例如，我们创建一个逐步传播的 Spiking ResNet-18[1]：

```
import torch
import torch.nn.functional as F
```

(续下页)

[1] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

(接上页)

```
from spikingjelly.clock_driven import neuron, surrogate, functional
from spikingjelly.clock_driven.model import spiking_resnet

net = spiking_resnet.spiking_resnet18(pretrained=False, progress=True, single_step_
↪neuron=neuron.IFNode, v_threshold=1., surrogate_function=surrogate.ATan())
print(net)
```

函数 `spiking_resnet18(pretrained=False, progress=True, single_step_neuron: callable=None, **kwargs)` 的运行参数中，`single_step_neuron` 是单步神经元，而 `**kwargs` 是神经元的参数。如果设置 `pretrained=True` 则可以加载 ANN，即 ResNet-18 的预训练模型参数。运行结果为：

```
SpikingResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),␣
↪bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
  (sn1): IFNode(
    v_threshold=1.0, v_reset=0.0, detach_reset=False
    (surrogate_function): ATan(alpha=2.0, spiking=True)
  )
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_
↪mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
    (1): BasicBlock(
```

(续下页)

(接上页)

```
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): BasicBlock(
```

(续下页)

(接上页)

```
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): BasicBlock(
```

(续下页)

(接上页)

```
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): BasicBlock(
```

(续下页)

(接上页)

```
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

单步网络的输入不包含时间维度，每次需要给网络单个时间步的输入：

```
net = spiking_resnet.spiking_resnet18(pretrained=False, progress=True, single_step_
↪neuron=neuron.IFNode, v_threshold=1., surrogate_function=surrogate.ATan())
T = 4
N = 2
x = torch.rand([T, N, 3, 224, 224])
fr = 0.
with torch.no_grad():
    for t in range(T):
        fr += net(x[t])
    fr /= T
print('firing rate =', fr)
```

搭建多步网络的方式类似，只需要把*spikingjelly.clock_driven.model.spiking_resnet.spiking_resnet18*换成*spikingjelly.clock_driven.model.spiking_resnet.multi_step_spiking_resnet18*并将单步神经元换成多步神经元：

```
net_ms = spiking_resnet.multi_step_spiking_resnet18(pretrained=False, progress=True,␣
↪multi_step_neuron=neuron.MultiStepIFNode, v_threshold=1., surrogate_
↪function=surrogate.ATan(), backend='torch')
print(net_ms)
```

运行结果为：

```
MultiStepSpikingResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),␣
↪bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
  (sn1): MultiStepIFNode(
    v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
    (surrogate_function): ATan(alpha=2.0, spiking=True)
  )
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_
↪mode=False)
  (layer1): Sequential(
    (0): MultiStepBasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
    (1): MultiStepBasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
```

(续下页)

(接上页)

```
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (layer2): Sequential(
    (0): MultiStepBasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): MultiStepBasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
```

(续下页)

(接上页)

```
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (layer3): Sequential(
    (0): MultiStepBasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): MultiStepBasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
```

(续下页)

(接上页)

```
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (layer4): Sequential(
    (0): MultiStepBasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): MultiStepBasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
```

(续下页)

(接上页)

```
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

对于多步网络，输入应该是带有时间维度的：

```
net = spiking_resnet.spiking_resnet18(pretrained=False, progress=True, single_step_
↪neuron=neuron.IFNode, v_threshold=1.,
                                      surrogate_function=surrogate.ATan())
T = 4
N = 2
x = torch.rand([T, N, 3, 224, 224])
fr = 0.
with torch.no_grad():
    for t in range(T):
        fr += net(x[t])
    fr /= T

net_ms = spiking_resnet.multi_step_spiking_resnet18(pretrained=False, progress=True,␣
↪multi_step_neuron=neuron.MultiStepIFNode, v_threshold=1., surrogate_
↪function=surrogate.ATan(), backend='torch')

net_ms.load_state_dict(net.state_dict())
with torch.no_grad():
    print('mse of single/multi step network outputs', F.mse_loss(net_ms(x).mean(0),␣
↪fr))
```

但是多步网络也允许输入不带时间维度的数据，在这种情况下，必须在构造网络时或构造后指定 `T`。

在构造时指定 `T`：

```
net_ms = spiking_resnet.multi_step_spiking_resnet18(pretrained=False, progress=True,␣
↪T=4, multi_step_neuron=neuron.MultiStepIFNode, v_threshold=1., surrogate_
↪function=surrogate.ATan(), backend='torch')
```

或者在构造后指定 `T`：

```
net_ms = spiking_resnet.multi_step_spiking_resnet18(pretrained=False, progress=True,␣
```

(续下页)

(接上页)

```
↪multi_step_neuron=neuron.MultiStepIFNode, v_threshold=1., surrogate_
↪function=surrogate.ATan(), backend='torch')
net_ms.T = 4
```

网络在 *forward* 时会将输入自动复制 `T` 次，和我们把输入复制是一样的：

```
net_ms = spiking_resnet.multi_step_spiking_resnet18(pretrained=False, progress=True,␣
↪multi_step_neuron=neuron.MultiStepIFNode, v_threshold=1., surrogate_
↪function=surrogate.ATan(), backend='torch')
T = 4
N = 2

with torch.no_grad():
    x = torch.rand([N, 3, 224, 224])
    y1 = net_ms(x.unsqueeze(0).repeat(T, 1, 1, 1, 1))
    functional.reset_net(net_ms)
    net_ms.T = T
    y2 = net_ms(x)
    print(F.mse_loss(y1, y2))
```

输出是：

```
tensor(0.)
```

让网络自行复制，计算效率会稍微高一些，原因参见*时间驱动：使用卷积 SNN 识别 Fashion-MNIST*。

1.18.2 在 ImageNet 上训练

ImageNet[2] 是计算机视觉常用的数据集，对于 SNN 而言颇具挑战性。惊蜇框架提供了一个训练 ImageNet 的代码样例，位于 spikingjelly.clock_driven.model.train_imagenet 。该代码样例的实现参考了 torchvision 。使用时只需要构建好网络、损失函数和正确率计算方式，就可以快速训练，下面是使用示例：

```
import torch
import torch.nn.functional as F
from spikingjelly.clock_driven.model import train_imagenet, spiking_resnet, train_
↪classify
from spikingjelly.clock_driven import neuron, surrogate

def ce_loss(x_seq: torch.Tensor, label: torch.Tensor):
    # x_seq.shape = [T, N, C]
    return F.cross_entropy(input=x_seq.mean(0), target=label)
```

(续下页)

[2] Deng, Jia, et al. "Imagenet: A large-scale hierarchical image database." 2009 IEEE conference on computer vision and pattern recognition. IEEE, 2009.

(接上页)

```
def cal_acc1_acc5(output, target):
    return train_classify.default_cal_acc1_acc5(output.mean(0), target)


if __name__ == '__main__':
    net = spiking_resnet.multi_step_spiking_resnet18(T=4, multi_step_neuron=neuron.
↪MultiStepIFNode, surrogate_function=surrogate.ATan(), detach_reset=True, backend=
↪'cupy')
    args = train_imagenet.parse_args()
    train_imagenet.main(model=net, criterion=ce_loss, args=args, cal_acc1_acc5=cal_
↪acc1_acc5)
```

我们把这段代码保存为 *resnet18_imagenet.py*。查看运行参数：

```
(pytorch-env) wfang@onebrain-dgx-a100-01:~/ssd/temp_dir$ python resnet18_imagenet.py -
↪h

                            [--step-gamma STEP_GAMMA] [--cosa-tmax COSA_TMAX] [--
↪momentum M] [--wd W] [--output-dir OUTPUT_DIR] [--resume RESUME] [--start-epoch N]␣
↪[--cache-dataset]
                            [--sync-bn] [--amp] [--world-size WORLD_SIZE] [--dist-url␣
↪DIST_URL] [--tb] [--T T] [--local_rank LOCAL_RANK]

PyTorch Classification Training

optional arguments:
  -h, --help            show this help message and exit
  --data-path DATA_PATH
                        dataset
  --device DEVICE       device
  -b BATCH_SIZE, --batch-size BATCH_SIZE
  --epochs N            number of total epochs to run
  -j N, --workers N     number of data loading workers (default: 16)
  --lr LR               initial learning rate
  --opt OPT             optimizer (sgd or adam)
  --lrs LRS             lr schedule (cosa(CosineAnnealingLR), step(StepLR)) or None
  --step-size STEP_SIZE
                        step_size for StepLR
  --step-gamma STEP_GAMMA
                        gamma for StepLR
  --cosa-tmax COSA_TMAX
                        T_max for CosineAnnealingLR. If none, it will be set to epochs
  --momentum M          Momentum for SGD
```

(续下页)

(接上页)

```
  --wd W, --weight-decay W
                        weight decay (default: 0)
  --output-dir OUTPUT_DIR
                        path where to save
  --resume RESUME       resume from checkpoint
  --start-epoch N       start epoch
  --cache-dataset       Cache the datasets for quicker initialization. It also␣
↪serializes the transforms
  --sync-bn             Use sync batch norm
  --amp                 Use AMP training
  --world-size WORLD_SIZE
                        number of distributed processes
  --dist-url DIST_URL   url used to set up distributed training
  --tb                  Use TensorBoard to record logs
  --T T                 simulation steps
  --local_rank LOCAL_RANK
```

在单卡上训练：

```
python resnet18_imagenet.py --data-path /raid/wfang/datasets/ImageNet --lr 0.1 --opt␣
↪sgd --lrs cosa --amp --tb --device cuda:7
```

在多卡上训练：

```
python -m torch.distributed.launch --nproc_per_node=8 resnet18_imagenet.py --data-
↪path /raid/wfang/datasets/ImageNet --lr 0.1 --opt sgd --lrs cosa --amp --tb
```

CHAPTER 2

模块文档

- *APIs*

CHAPTER 3

文档索引

- genindex
- modindex
- search

CHAPTER 4

引用和出版物

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},
    howpublished = {\url{https://github.com/fangwei123456/spikingjelly}},
    note = {Accessed: YYYY-MM-DD},
}
```

其中的 *YYYY-MM-DD* 需要更改为您的工作使用的惊蜇 (SpikingJelly) 版本对应的最后一次代码修改日期。

使用惊蜇 (SpikingJelly) 的出版物可见于 Publications using 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

友情链接

- 脉冲神经网络相关博客
- 脉冲强化学习相关博客

CHAPTER 7

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

7.1 Installation

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

The odd version number is the developing version, which is updated with GitHub/OpenI repository. The even version number is the stable version and available at PyPI.

Install the last stable version from PyPI:

```
pip install spikingjelly
```

Install the latest developing version 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
```

7.1.1 Clock_driven

Author: fangwei123456, lucifer2859

This tutorial focuses on `spikingjelly.clock_driven`, introducing the clock-driven simulation method, the concept of surrogate gradient method, and the use of differentiable spiking neurons.

The surrogate gradient method is a new method emerging in recent years. For more information about this method, please refer to the following overview:

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.

The download address for this article can be found at arXiv .

SNN Compared with RNN

The neuron in SNN can be regarded as a kind of RNN, and its input is the voltage increment (or the product of current and membrane resistance, but for convenience, `clock_driven.neuron` uses voltage increment). The hidden state is the membrane voltage, and the output is a spike. Such spiking neurons are Markovian: the output at the current time is only related to the input at the current time and the state of the neuron itself.

You can use three discrete equations ——Charge, Discharge, Reset ——to describe any discrete spiking neuron:

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

where $V(t)$ is the membrane voltage of the neuron; $X(t)$ is an external source input, such as voltage increment; $H(t)$ is the hidden state of the neuron, which can be understood as the instant before the neuron has not fired a spike; $f(V(t-1), X(t))$ is the state update equation of the neuron. Different neurons differ in the update equation.

For example, for a LIF neuron, we describe the differential equation of its dynamics below a threshold, and the corresponding difference equation are:

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

The corresponding Charge equation is

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

In the Discharge equation, $S(t)$ is a spike fired by a neuron, $g(x) = \Theta(x)$ is a step function. RNN is used to call it a gating function. In SNN, it is called a spiking function. The output of the spiking function is only 0 or 1, which can represent the firing process of spike, defined as

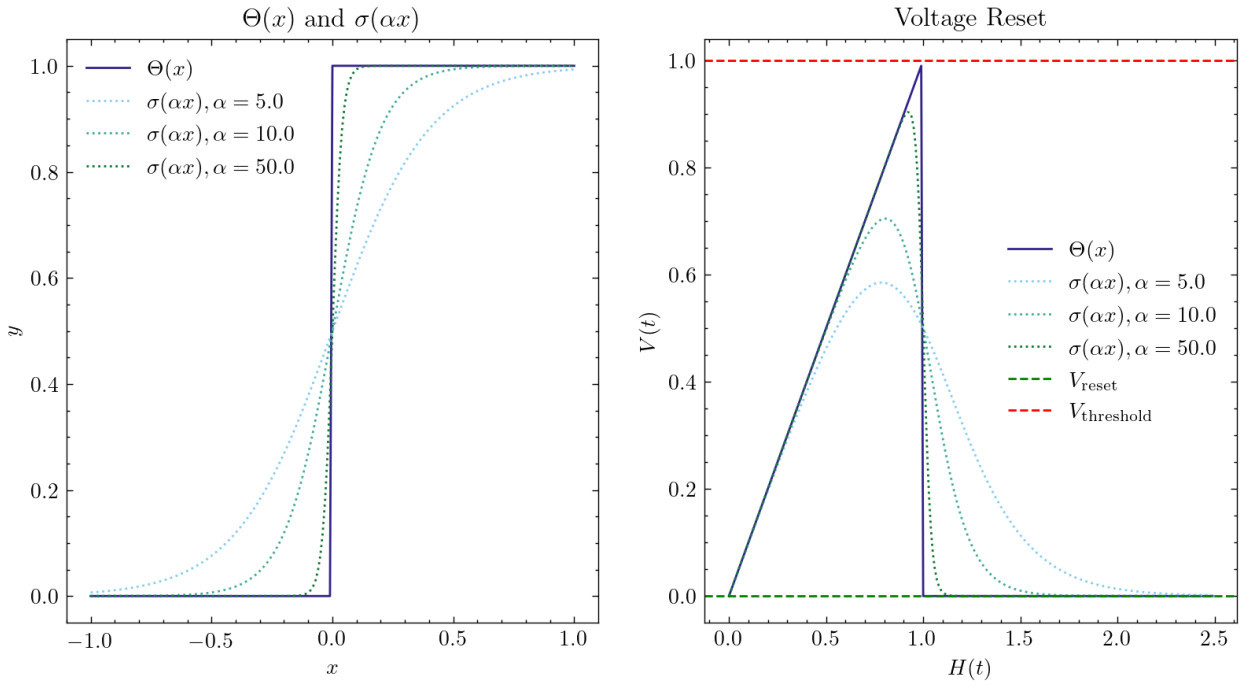
$$\Theta(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Reset means the reset process of the voltage: when a spike is fired, the voltage is reset to V_{reset}; If no spike is fired, the voltage remains unchanged.

Surrogate Gradient Method

RNN uses differentiable gating functions, such as the tanh function. Obviously, the spiking function of SNN $g(x) = \Theta(x)$ is not differentiable, which leads to the fact that SNN is very similar to RNN to a certain extent, but it cannot be trained by gradient descent and back-propagation. We can use a gating function that is very similar to $g(x) = \Theta(x)$, but differentiable $\sigma(x)$ to replace it.

The core idea of this method is: when forwarding, using $g(x) = \Theta(x)$, the output of the neuron is discrete 0 and 1, and our network is still SNN; When back-propagation, the gradient of the surrogate gradient function $g'(x) = \sigma'(x)$ is used to replace the gradient of the spiking function. The most common surrogate gradient function is the sigmoid function $\sigma(\alpha x) = \frac{1}{1+exp(-\alpha x)}$. α can control the smoothness of the function. The function with larger α will be closer to $\Theta(x)$. But when it gets closer to $x = 0$, the gradient will be more likely to explode. And when it gets farther to $x = 0$, the gradient will be more likely to disappear. This makes the network more difficult to train. The following figure shows the shape of the surrogate gradient function and the corresponding Reset equation for different α:



The default surrogate gradient function is `clock_driven.surrogate.Sigmoid()`, `clock_driven.surrogate` also provides other optional approximate gating functions. The surrogate gradient function is one of the

parameters of the neuron constructor in `clock_driven.neuron`:

```
class BaseNode(base.MemoryModule):
    def __init__(self, v_threshold: float = 1., v_reset: float = 0.,
                 surrogate_function: Callable = surrogate.Sigmoid(), detach_reset:␣
↪bool = False):
        """
        :param v_threshold: threshold voltage of neurons
        :type v_threshold: float

        :param 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``
        :type v_reset: float

        :param surrogate_function: surrogate function for replacing gradient of␣
↪spiking functions during back-propagation
        :type surrogate_function: Callable

        :param detach_reset: whether detach the computation graph of reset
        :type detach_reset: bool

        This class is the base class of differentiable spiking neurons.
        """
```

If you want to customize the new approximate gating function, you can refer to the code in `clock_driven.surrogate`. Usually we define it as `torch.autograd.Function`, and then encapsulate it into a subclass of `torch.nn.Module`.

Embed Spiking Neurons into Deep Networks

After solving the differential problem of spiking neurons, our spiking neurons can be embedded into any network built using PyTorch like an activation function, making the network an SNN. Some classic neurons have been implemented in `clock_driven.neuron`, which can easily build various networks, such as a simple fully connected network:

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

Example: MNIST classification using a single-layer fully connected network

Now we use the LIF neurons in `clock_driven.neuron` to build a one-layer fully connected network to classify the MNIST dataset.

Firstly, we confirm hyperparameters we needed:

```
parser.add_argument('--device', default='cuda:0', help='Device, e.g., "cpu" or "cuda:0
↪"')

parser.add_argument('--dataset-dir', default='./', help='Root directory for saving␣
↪MNIST dataset, e.g., "./"')
parser.add_argument('--log-dir', default='./', help='Root directory for saving␣
↪tensorboard logs, e.g., "./"')
parser.add_argument('--model-output-dir', default='./', help='Model directory for␣
↪saving, e.g., "./"')

parser.add_argument('-b', '--batch-size', default=64, type=int, help='Batch size, e.g.
↪, "64"')
parser.add_argument('-T', '--timesteps', default=100, type=int, dest='T', help=
↪'Simulating timesteps, e.g., "100"')
parser.add_argument('--lr', '--learning-rate', default=1e-3, type=float, metavar='LR',
↪ help='Learning rate, e.g., "1e-3": ', dest='lr')
parser.add_argument('--tau', default=2.0, type=float, help='Membrane time constant,␣
↪tau, for LIF neurons, e.g., "100.0"')
parser.add_argument('-N', '--epoch', default=100, type=int, help='Training epoch, e.g.
↪, "100"')
```

Initialize the DataLoader:

```
# Initialize the DataLoader
train_dataset = torchvision.datasets.MNIST(
    root=dataset_dir,
    train=True,
    transform=torchvision.transforms.ToTensor(),
    download=True
)
test_dataset = torchvision.datasets.MNIST(
    root=dataset_dir,
    train=False,
    transform=torchvision.transforms.ToTensor(),
    download=True
)

train_data_loader = data.DataLoader(
```

(续下页)

(接上页)

```
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True,
    drop_last=True
)
test_data_loader = data.DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    shuffle=False,
    drop_last=False
)
```

Define our network structure:

```
# Define SNN
net = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 10, bias=False),
    neuron.LIFNode(tau=tau)
)
net = net.to(device)
```

Initialize the optimizer and encoder (we use a Poisson encoder to encode the MNIST image into spike trains):

```
# Use Adam optimizer
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
# Use Poisson encoder
encoder = encoding.PoissonEncoder()
```

The training of the network is simple. Run the network for `T` time steps to accumulate the output spikes of 10 neurons in the output layer to obtain the number of spikes fired by the output layer `out_spikes_counter`; Use the firing times of the spike divided by the simulation duration to get the firing frequency of the output layer `out_spikes_counter_frequency = out_spikes_counter / T`. We hope that when the real category of the input image is `i`, the `i`-th neuron in the output layer has the maximum activation degree, while the other neurons remain silent. Therefore, the loss function is naturally defined as the firing frequency of the output layer `out_spikes_counter_frequency` and the cross-entropy of `label_one_hot` obtained after one-hot encoding with the real category, or MSE. We use MSE because the experiment found that MSE is better. In particular, note that SNN is a stateful, or memorized network. So before entering new data, you must reset the state of the network. This can be done by calling `clock_driven.functional.reset_net(net)` to fulfill. The training code is as follows:

```
print("Epoch {}:".format(epoch))
print("Training...")
train_correct_sum = 0
```

(续下页)

(接上页)

```
train_sum = 0
net.train()
for img, label in tqdm(train_data_loader):
    img = img.to(device)
    label = label.to(device)
    label_one_hot = F.one_hot(label, 10).float()

    optimizer.zero_grad()

    # Run for T durations, out_spikes_counter is a tensor with shape=[batch_size, 10]
    # Record the number of spikes delivered by the 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 firing frequency of 10 neurons in the␣
↪output layer within the simulation duration
    out_spikes_counter_frequency = out_spikes_counter / T

    # The loss function is the firing frequency of the neurons in the output layer,␣
↪and the MSE of the real class
    # Such a loss function causes that when the category i is input, the firing␣
↪frequency of the i-th neuron in the output layer approaches 1, while the firing␣
↪frequency of other neurons approaches 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 SNN neurons have "memory"
    functional.reset_net(net)

    # Calculation of accuracy. The index of the neuron with max frequency in the␣
↪output layer is the classification result.
    train_correct_sum += (out_spikes_counter_frequency.max(1)[1] == label.to(device)).
↪float().sum().item()
    train_sum += label.numel()

    train_batch_accuracy = (out_spikes_counter_frequency.max(1)[1] == label.
↪to(device)).float().mean().item()
    writer.add_scalar('train_batch_accuracy', train_batch_accuracy, train_times)
```

(续下页)

(接上页)

```
    train_accs.append(train_batch_accuracy)

    train_times += 1
train_accuracy = train_correct_sum / train_sum
```

The test code is simpler than the training code:

```
print("Testing...")
net.eval()
with torch.no_grad():
    # Each time through the entire data set, test once on the test set
    test_sum = 0
    correct_sum = 0
    for img, label in tqdm(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)
    test_accuracy = correct_sum / test_sum
    writer.add_scalar('test_accuracy', test_accuracy, epoch)
    test_accs.append(test_accuracy)
    max_test_accuracy = max(max_test_accuracy, test_accuracy)
print("Epoch {}: train_acc={}, test_acc={}, max_test_acc={}, train_times={}".
↪format(epoch, train_accuracy, test_accuracy, max_test_accuracy, train_times))
print()
```

The complete code is located at `clock_driven.examples.lif_fc_mnist.py`. In the code, we also use Tensorboard to save the training log. Here are the (hyper)parameters you can configure:

```
$ python <PATH>/lif_fc_mnist.py --help
usage: lif_fc_mnist.py [-h] [--device DEVICE] [--dataset-dir DATASET_DIR] [--log-dir␣
↪LOG_DIR] [--model-output-dir MODEL_OUTPUT_DIR] [-b BATCH_SIZE] [-T T] [--lr LR] [--
↪tau TAU] [-N EPOCH]

spikingjelly LIF MNIST Training

optional arguments:
```

(续下页)

(接上页)

```
-h, --help            show this help message and exit
--device DEVICE       运行的设备，例如“cpu”或“cuda:0” Device, e.g., "cpu" or
↪"cuda:0"
--dataset-dir DATASET_DIR
                      保存MNIST数据集的位置，例如“./” Root directory for saving␣
↪MNIST dataset, e.g., "./"
--log-dir LOG_DIR     保存tensorboard日志文件的位置，例如“./” Root directory for␣
↪saving tensorboard logs, e.g., "./"
--model-output-dir MODEL_OUTPUT_DIR
                      模型保存路径，例如“./” Model directory for saving, e.g., "./
↪"
-b BATCH_SIZE, --batch-size BATCH_SIZE
                      Batch 大小，例如“64” Batch size, e.g., "64"
-T T, --timesteps T   仿真时长，例如“100” Simulating timesteps, e.g., "100"
--lr LR, --learning-rate LR
                      学习率，例如“1e-3” Learning rate, e.g., "1e-3":
--tau TAU             LIF神经元的时间常数tau，例如“100.0” Membrane time constant,␣
↪tau, for LIF neurons, e.g., "100.0"
-N EPOCH, --epoch EPOCH
                      训练epoch，例如“100” Training epoch, e.g., "100"
```

You can also run it directly on the Python command line:

```
$ python
>>> import spikingjelly.clock_driven.examples.lif_fc_mnist as lif_fc_mnist
>>> lif_fc_mnist.main()
########## Configurations ##########
device=cuda:0
dataset_dir=./
log_dir=./
model_output_dir=./
batch_size=64
T=100
lr=0.001
tau=2.0
epoch=100
####################################
Epoch 0:
Training...
100
↪%|████████████████████████████████████████████████████████████████████████████████
↪937/937 [01:26<00:00, 10.89it/s]
Testing...
100
```

(续下页)

(接上页)

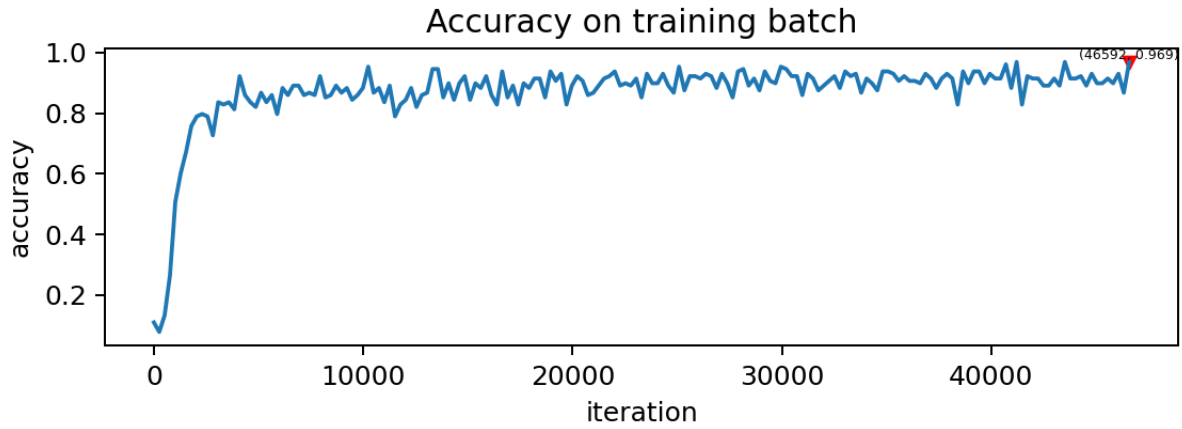
```
↪%|
↪157/157 [00:05<00:00, 28.79it/s]
Epoch 0: train_acc = 0.8641775613660619, test_acc=0.9071, max_test_acc=0.9071, train_
↪times=937
```

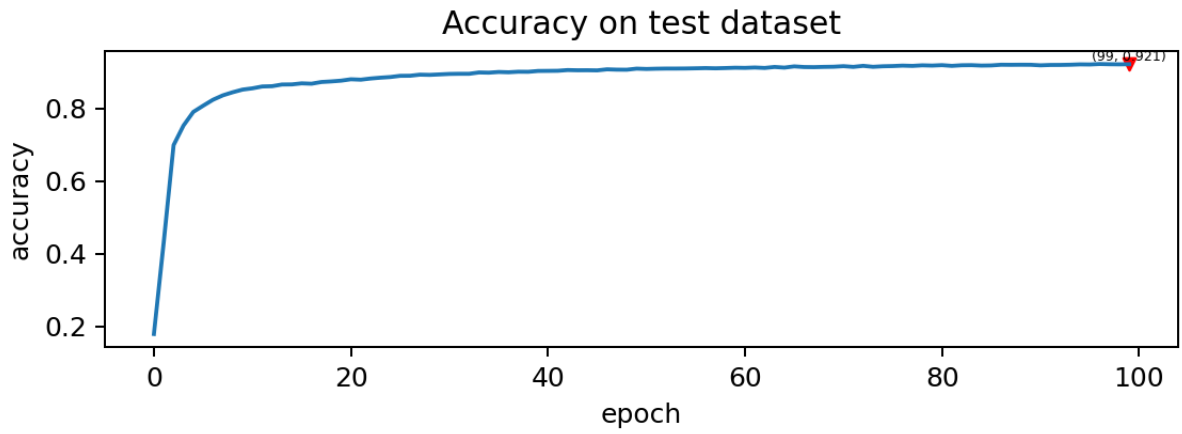
Save and load model：

```
# Save model
torch.save(net, model_output_dir + "/lif_snn_mnist.ckpt")
# Load model
# net = torch.load(model_output_dir + "/lif_snn_mnist.ckpt")
```

It should be noted that the amount of memory required to train such an SNN is linearly related to the simulation time `T`. A longer `T` is equivalent to using a smaller simulation step size and training is more “fine”, however, the training effect is not necessarily better. So if `T` is too large, the SNN will become a very deep network after being expanded in time, and the gradient is easy to decay or explode. Since we use a Poisson encoder, a larger `T` is required.

Our model, training 100 epochs on Tesla K80, takes about 75 minutes. The changes in the accuracy of each batch and the accuracy of the test set during training are as follows:





The final test set accuracy rate is about 92%, which is not a very high accuracy rate, because we use a very simple network structure and Poisson encoder. We can completely remove the Poisson encoder and send the image directly to the SNN. In this case, the first layer of LIF neurons can be regarded as an encoder.

7.1.2 Clock driven: Neurons

Author: fangwei123456

Translator: YeYumin

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

Spiking Nneuron Model

In `spikingjelly`, we define the 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 (SNNs). *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 we create a new LIF neurons layer:

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

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

- **tau** –membrane time constant
- **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

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. We can just ignore it now.

You may be curious about the number of neurons in this layer. For most neurons 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.

Similar to neurons in RNN, spiking neurons are also stateful (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`. We can print the membrane potential of the newly created LIF neurons layer:

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

We can find that `lif.v` is now `0.0` because we haven't given it any input yet. We give several different inputs and observe the `shape` of `lif.v`. We can find that it is consistent with the numel of inputs:

```
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()
```

What is the relationship between V_t and input X_t? In the spiking neuron, it not only depends on the input X_t at time-step `t`, but also on its membrane potential V_{t-1} at the last time-step `t-1`.

We often use the sub-threshold (when the membrane potential does not exceed the threshold potential `V_{threshold}`) neuronal dynamics equation $\frac{\mathrm{d}V(t)}{\mathrm{d}t} = f(V(t), X(t))$ to describe the continuous-time spiking neuron. For example. For LIF neurons, the equation is:

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

where τ_m is the membrane time constant and V_{reset} is the reset potential. For such a differential equation, $X(t)$ is not a constant and 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 discrete equation, the charging equation of the LIF neuron is:

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

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 *spikingjelly.clock_driven.neuron.LIFNode.neuronal_charge*:

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

    else:
        if isinstance(self.v_reset, float) and self.v_reset == 0.:
            self.v += (x - self.v) / self.tau
        else:
            self.v += (x - (self.v - self.v_reset)) / self.tau
```

Different neurons have different charging equations. However, when the membrane potential exceeds the threshold potential, the release of spike and the reset of the membrane potential are the same for all kinds of neurons. Therefore, they all inherit from *spikingjelly.clock_driven.neuron.BaseNode* and share the same discharge and reset equations. The codes of neuronal fire can be found at *spikingjelly.clock_driven.neuron.BaseNode.neuronal_fire*:

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

`surrogate_function()` is a heaviside step function during forward propagation. When 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 spikes consumes the previously accumulated electric charge of the neuron, so there will be an instantaneous decrease in the membrane potential, which is the neuronal reset. In SNNs, there are two ways to realize neuronal reset:

1. Hard method: After releasing a spike, the membrane potential is directly set to the reset potential $V = V_{reset}$
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 neurons in *spikingjelly.clock_driven.neuron*, when `v_reset` is set to the a float value (e.g., the default value is `1.0`), the neuron uses the hard reset; if `v_reset` is set to `None`, the soft reset will be used. We can find the corresponding codes in `spikingjelly.clock_driven.neuron.BaseNode.neuronal_fire.neuronal_reset`:

```
def neuronal_reset(self):
    # ...
    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
```

Three Equations to Describe Discrete Spiking Neurons

We can use the three discrete equations: neuronal charge, neuronal fire, and neuronal reset to describe all kinds of discrete spiking neurons. The neuronal charge and fire equations are:

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

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

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

The hard reset is:

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

The soft reset 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 after neuronal charge but before neuronal fire, V_t is the membrane potential after the neuronal fire, $f(V(t-1), X(t))$ is the neuronal charge function. The difference between neurons is the neuronal charge.

Clock-driven Simulation

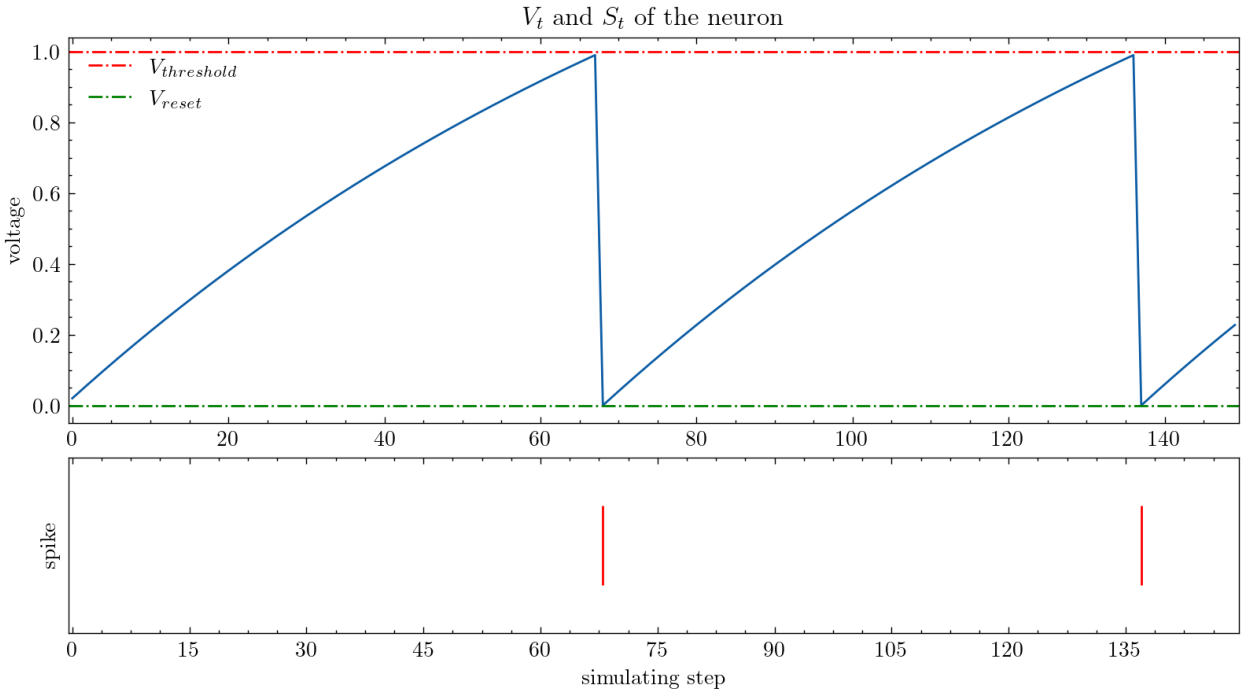
spikingjelly.clock_driven uses a clock-driven approach to simulate SNN.

Next, we will stimulate the neuron and check its membrane potential and output spikes.

Now let us give constant input to the LIF neurons layer and plot the membrane potential and output spikes:

```
lif.reset()
x = torch.as_tensor([2.])
T = 150
s_list = []
v_list = []
for t in range(T):
    s_list.append(lif(x))
    v_list.append(lif.v)

visualizing.plot_one_neuron_v_s(np.asarray(v_list), np.asarray(s_list), v_
↪threshold=lif.v_threshold, v_reset=lif.v_reset,
                                dpi=200)
plt.show()
```

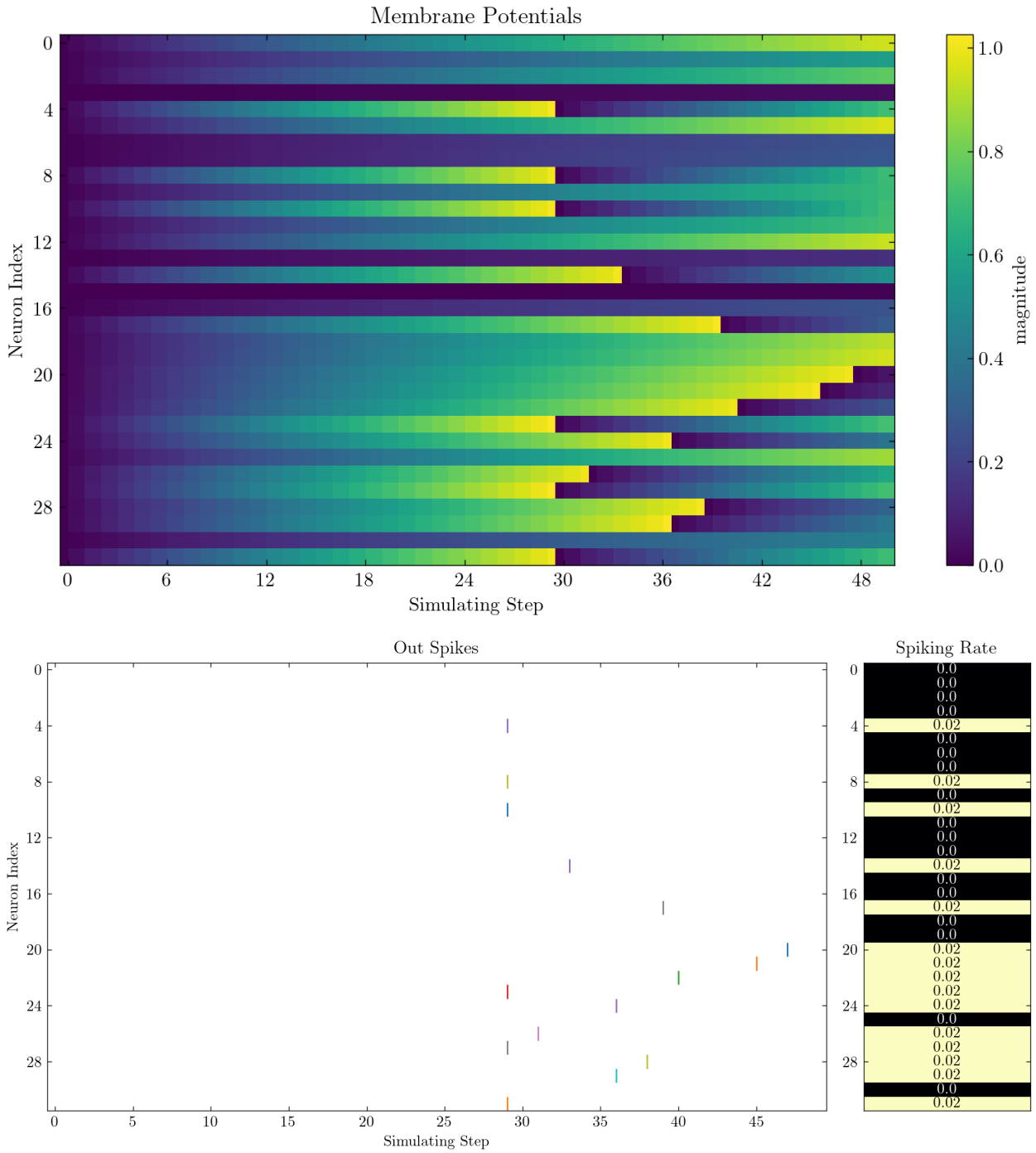
The input is with `shape=[1]`, and this LIF neurons layer has only 1 neuron. Its membrane potential and output spikes change with time-step as follows:



We reset the neurons layer and give an input with `shape=[32]` to see the membrane potential and output spikes of these 32 neurons:

```
lif.reset()
x = torch.rand(size=[32]) * 4
T = 50
s_list = []
v_list = []
for t in range(T):
    s_list.append(lif(x).unsqueeze(0))
    v_list.append(lif.v.unsqueeze(0))

s_list = torch.cat(s_list)
v_list = torch.cat(v_list)

visualizing.plot_2d_heatmap(array=np.asarray(v_list), 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(s_list), title='Membrane Potentials',␣
↪xlabel='Simulating Step',
                           ylabel='Neuron Index', dpi=200)
plt.show()
```

The results are as follows:


Membrane Potentials
Neuron Index
Simulating Step
magnitude
Out Spikes
Spiking Rate
Neuron Index
Simulating Step

7.1.3 Clock driven: Encoder

Author: Grasshlw, Yanqi-Chen, fangwei123456

Translator: YeYumin

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

The Base Class of Encoder

All encodes are based on two base encoders:

1. The stateless base encoder *spikingjelly.clock_driven.encoding.StatelessEncoder*
2. The stateful base encoder *spikingjelly.clock_driven.encoding.StatefulEncoder*

There are no hidden states in the stateless encoder, and the spikes `spike[t]` will be encoded from the input data `x[t]` at time-step `t`. While the stateful encoder `encoder = StatefulEncoder(T)` will use `encode` function to encode the input sequence `x` containing `T` time-steps data to `spike` at the first time of `forward`, and will output `spike[t % T]` at the"t"-th calling `forward`. The codes of *spikingjelly.clock_driven.encoding.StatefulEncoder.forward* are:

```
def forward(self, x: torch.Tensor):
    if self.spike is None:
        self.encode(x)

    t = self.t
    self.t += 1
    if self.t >= self.T:
        self.t = 0
    return self.spike[t]
```

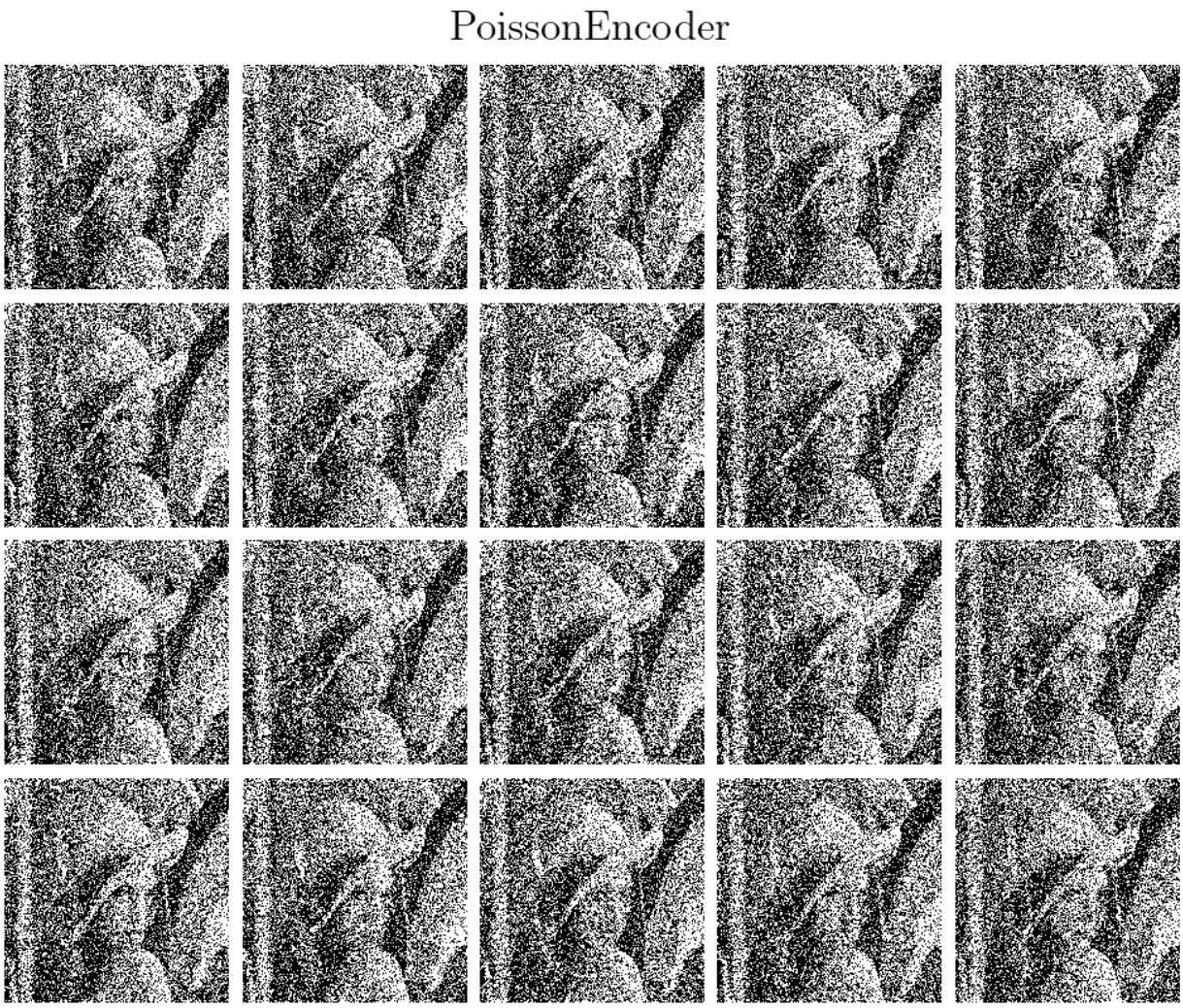
Poisson Encoder

The Poisson encoder *spikingjelly.clock_driven.encoding.PoissonEncoder* is a stateless encoder. It converts the input data `x` into a spike with the same shape, 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

# 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.imshow(x, cmap='gray')
plt.axis('off')

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

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:

```
#␣
↪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
```

(续下页)

(接上页)

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

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.

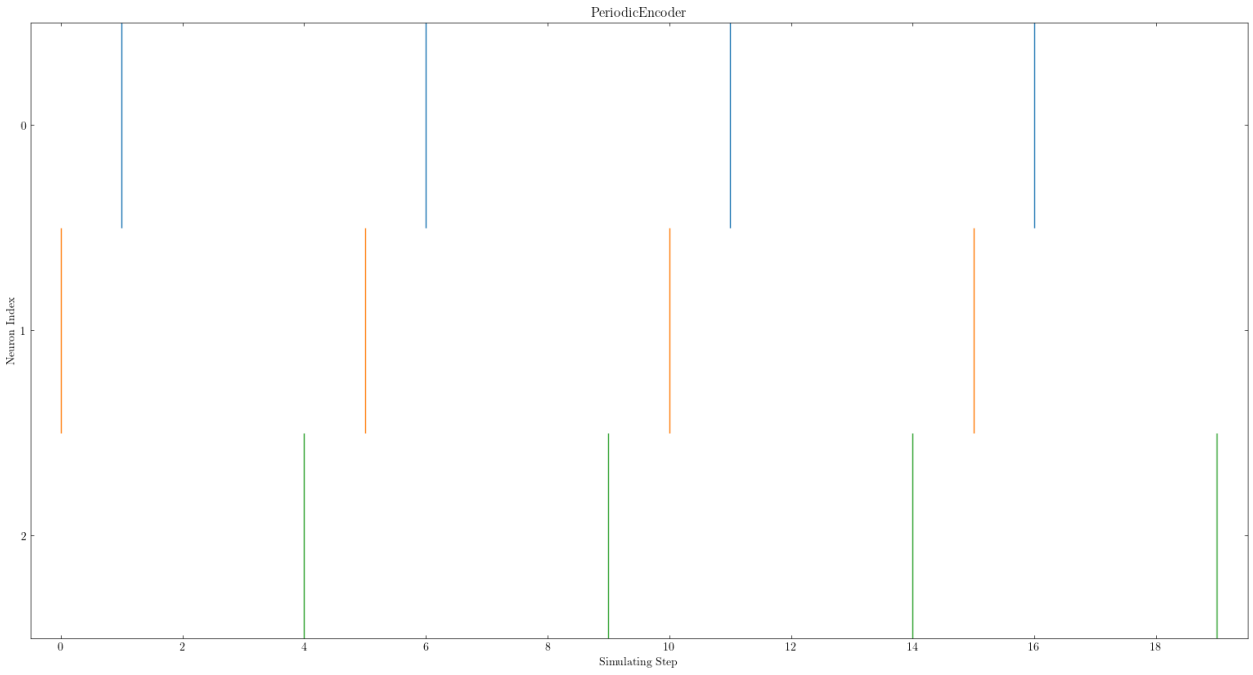
Periodic Encoder

Periodic encoder *spikingjelly.clock_driven.encoding.PoissonEncoder* is an encoder that periodically outputs spikes from a given spike sequence. `spike` is set at the initialization of `PeriodicEncoder`, and we can also use `spikingjelly.clock_driven.encoding.PoissonEncoder.encode` to set a new `spike`.

```
class PeriodicEncoder(BaseEncoder):
    def __init__(self, spike: torch.Tensor):
        super().__init__(spike.shape[0])
        self.encode(spike)
    def encode(self, spike: torch.Tensor):
        self.spike = spike
        self.T = spike.shape[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.

```
spike = torch.full((5, 3), 0)
spike[1, 0] = 1
spike[0, 1] = 1
spike[4, 2] = 1

pe = encoding.PeriodicEncoder(spike)

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

visualizing.plot_1d_spikes(out_spike.float().numpy(), 'PeriodicEncoder', 'Simulating␣
↪Step', 'Neuron Index',
                           plot_firing_rate=False)
plt.show()
```



Latency encoder

The latency encoder *spikingjelly.clock_driven.encoding.LatencyEncoder* 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_f and the stimulus intensity $x \in [0, 1]$ satisfy the following formulas. When the encoding type is

linear (`function_type='linear'`)

$$t_f(x) = (T-1)(1-x)$$

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:

$$(T-1) - ln(\alpha * 1 + 1) = 0$$

This may cause the encoder to overflow:

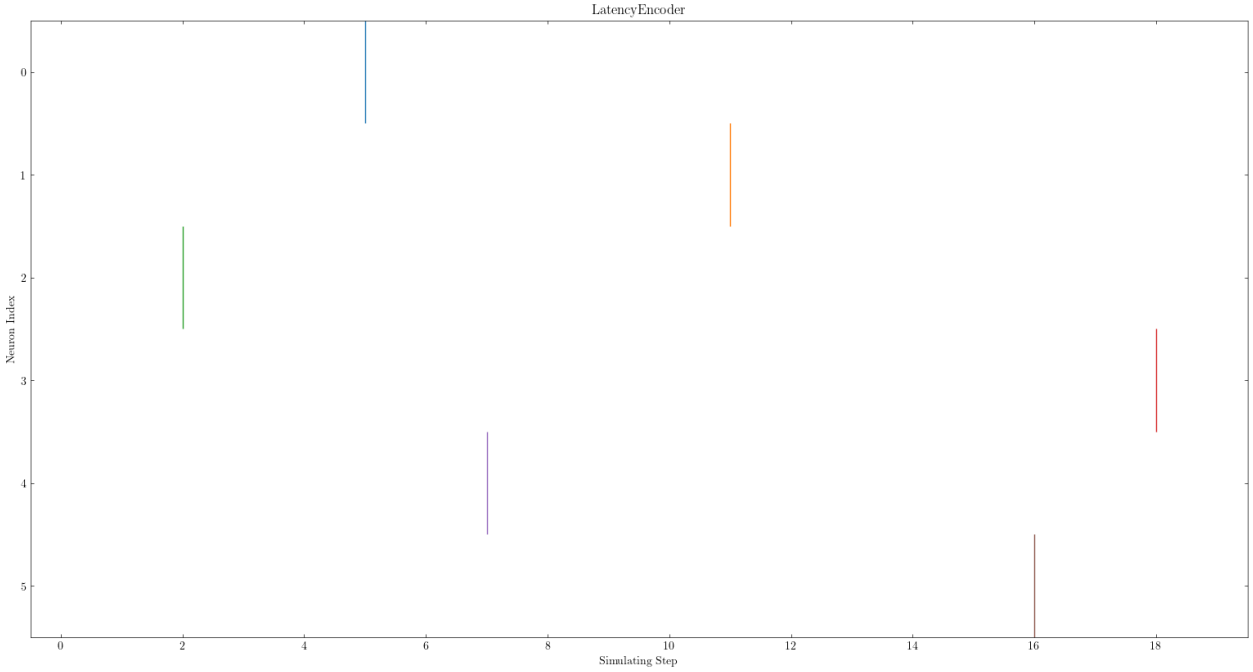
$$\alpha = e^{T-1} - 1$$

because α will increase exponentially as T 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.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
N = 6
x = torch.rand([N])
T = 20

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

# 输出延迟编码器的编码结果
out_spike = torch.zeros([T, N])
for t in range(T):
    out_spike[t] = le(x)

print(x)
visualizing.plot_1d_spikes(out_spike.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:



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

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

7.1.4 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:

```
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 such as learning rate and several other configurations

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=lr)
# 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:

```
print("Epoch {}:".format(epoch))
print("Training...")
train_correct_sum = 0
train_sum = 0
net.train()
for img, label in tqdm(train_data_loader):
    img = img.to(device)
    label = label.to(device)
    label_one_hot = F.one_hot(label, 10).float()

    optimizer.zero_grad()

    # Run for T durations, out_spikes_counter is a tensor with shape=[batch_size, 10]
    # Record the number of spikes delivered by the 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 firing frequency of 10 neurons in the␣
↪output layer within the simulation duration
    out_spikes_counter_frequency = out_spikes_counter / T

    # The loss function is the firing frequency of the neurons in the output layer,␣
↪and the MSE of the real class
    # Such a loss function causes that when the category i is input, the firing␣
↪frequency of the i-th neuron in the output layer approaches 1, while the firing␣
↪frequency of other neurons approaches 0.
```

(续下页)

(接上页)

```
    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 SNN neurons have "memory"
    functional.reset_net(net)

    # Calculation of accuracy. The index of the neuron with max frequency in the␣
↪output layer is the classification result.
    train_correct_sum += (out_spikes_counter_frequency.max(1)[1] == label.to(device)).
↪float().sum().item()
    train_sum += label.numel()

    train_batch_accuracy = (out_spikes_counter_frequency.max(1)[1] == label.
↪to(device)).float().mean().item()
    writer.add_scalar('train_batch_accuracy', train_batch_accuracy, train_times)
    train_accs.append(train_batch_accuracy)

    train_times += 1
train_accuracy = train_correct_sum / train_sum
```

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 command line:

```
$ python <PATH>/lif_fc_mnist.py --help
usage: lif_fc_mnist.py [-h] [--device DEVICE] [--dataset-dir DATASET_DIR] [--log-dir␣
↪LOG_DIR] [-b BATCH_SIZE] [-T T] [--lr LR] [--gpu GPU]
                       [--tau TAU] [-N EPOCH]

spikingjelly MNIST Training

optional arguments:
-h, --help            show this help message and exit
--device DEVICE       运行的设备，例如“cpu”或“cuda:0” Device, e.g., "cpu" or
↪"cuda:0"
--dataset-dir DATASET_DIR
                        保存MNIST数据集的位置，例如“./” Root directory for saving␣
↪MNIST dataset, e.g., "./"
--log-dir LOG_DIR     保存tensorboard日志文件的位置，例如“./” Root directory for␣
↪saving tensorboard logs, e.g., "./"
-b BATCH_SIZE, --batch-size BATCH_SIZE
-T T, --timesteps T   仿真时长，例如“100” Simulating timesteps, e.g., "100"
--lr LR, --learning-rate LR
```

(续下页)

(接上页)

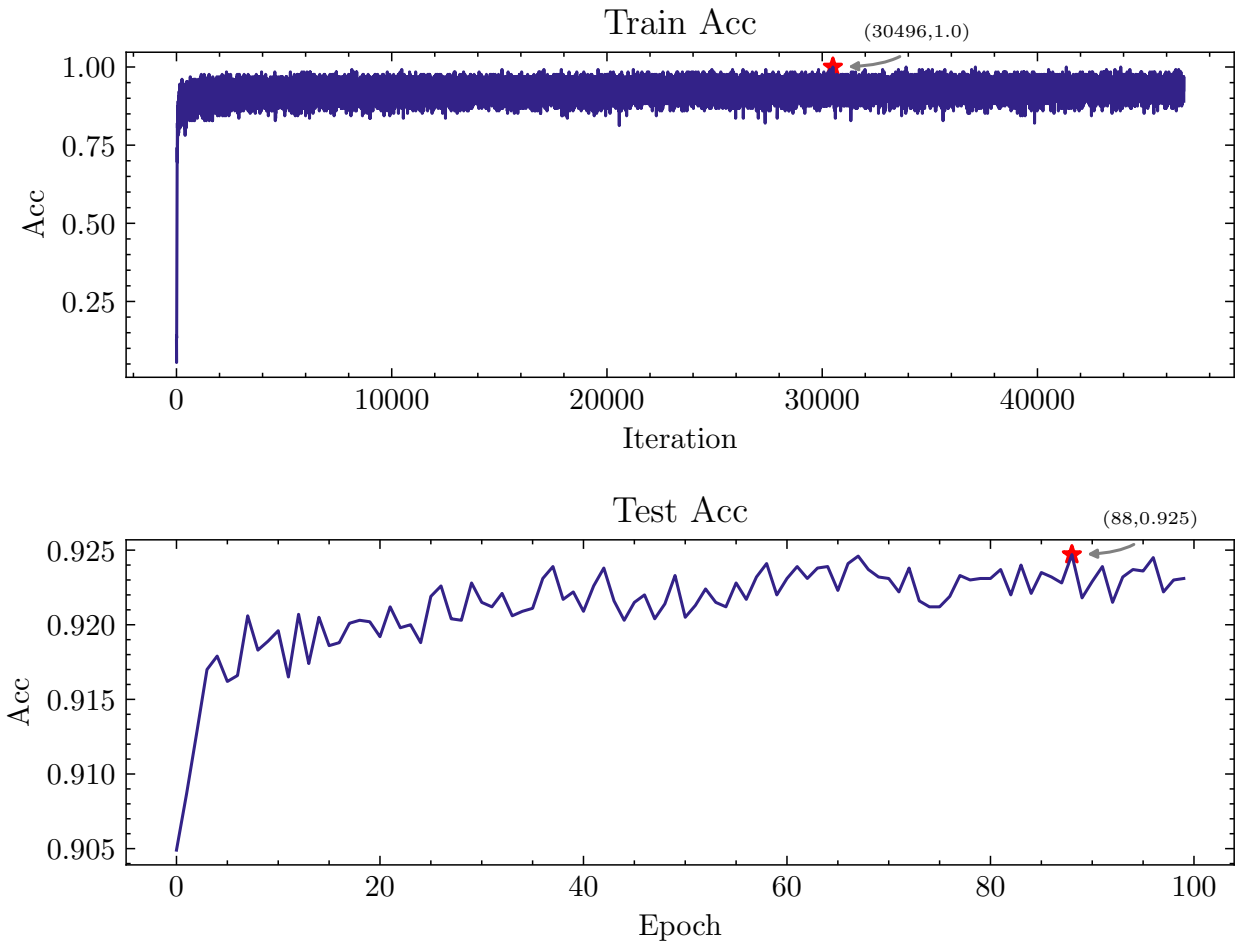
```
                        学习率，例如“1e-3” Learning rate, e.g., "1e-3":
--gpu GPU               GPU id to use.
--tau TAU               LIF神经元的时间常数tau，例如“100.0” Membrane time constant,␣
↪tau, for LIF neurons, e.g., "100.0"
-N EPOCH, --epoch EPOCH
```

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.

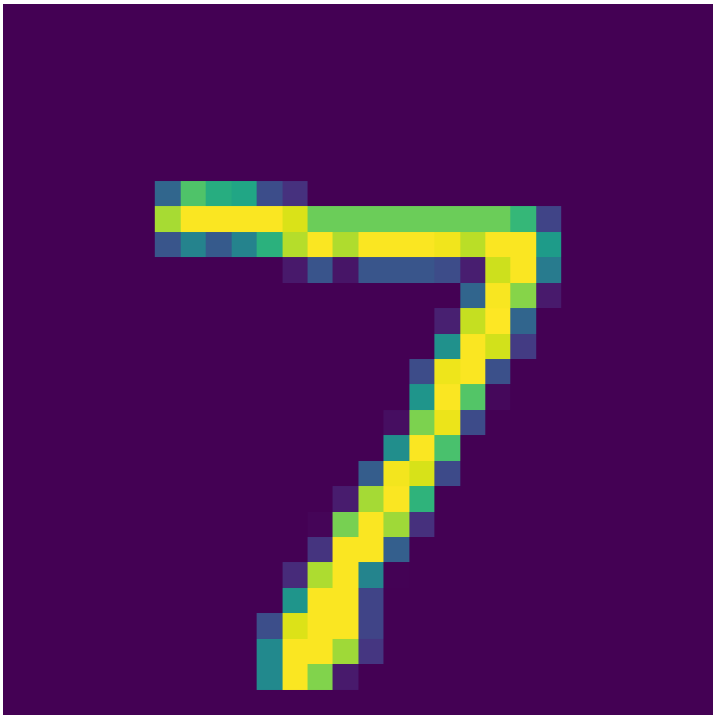
In addition, because we use a Poisson encoder, a larger `T` is required.

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



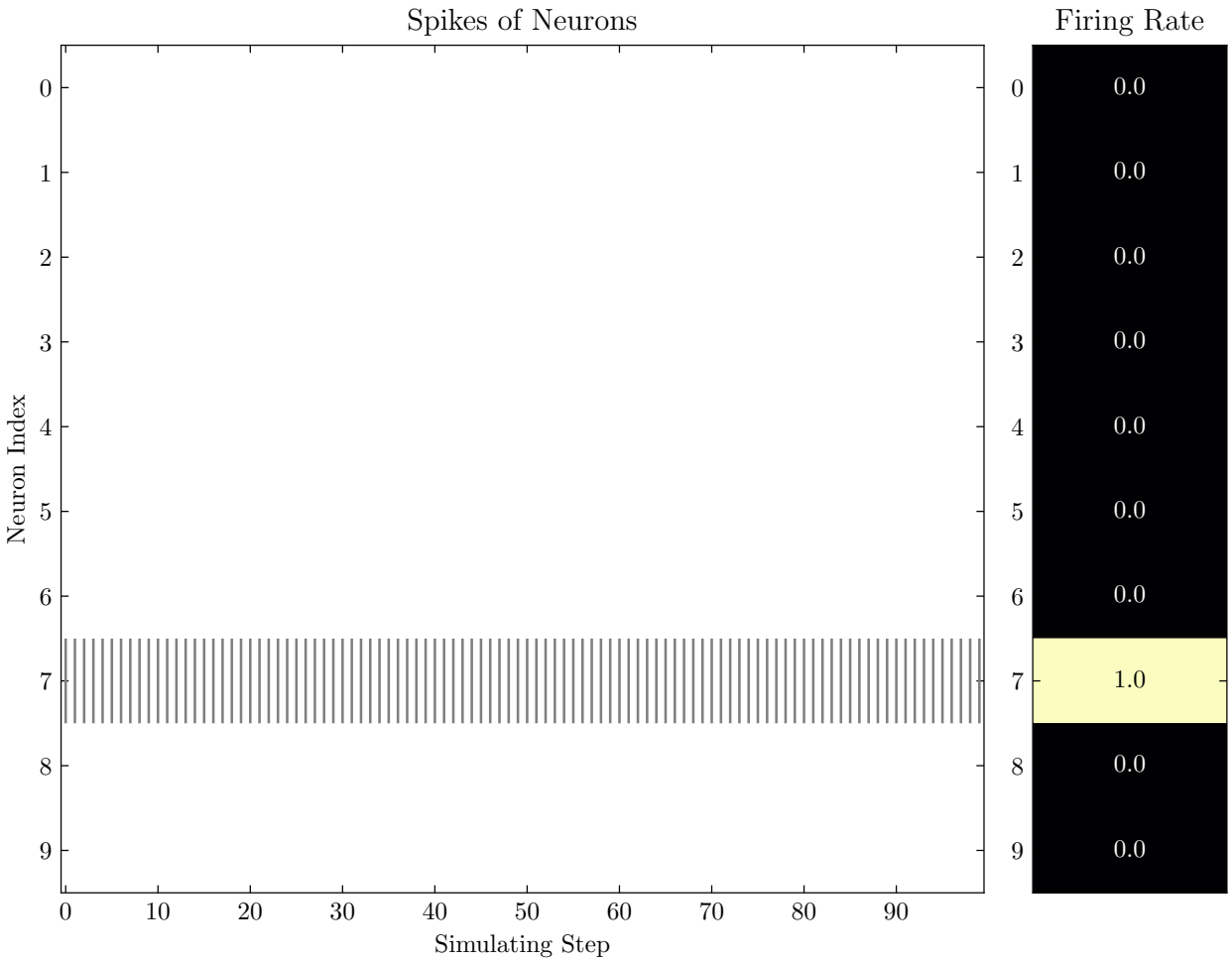
Select the first picture in the test set:

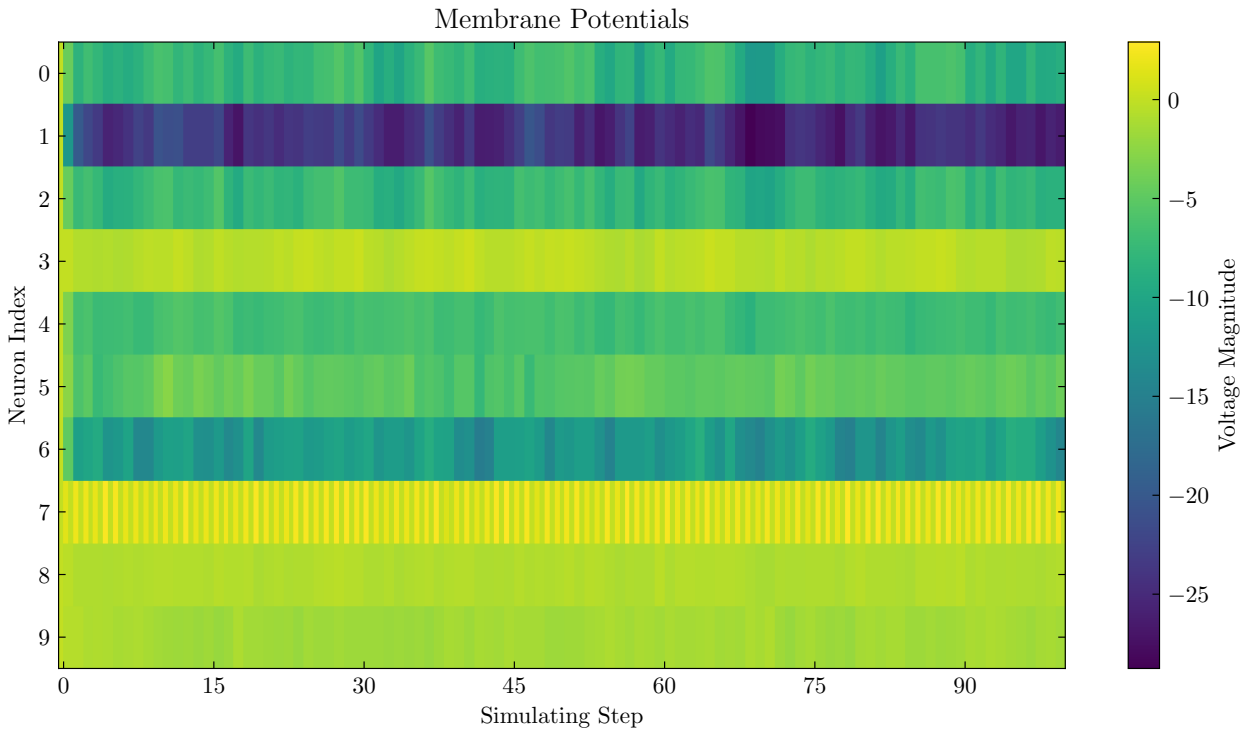


Use the trained model to classify and get the classification result.

```
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
0.0
0.0
0.0
0.0
0.0
0.0
1.0
0.0
0.0
0
10
20
30
40
50
60
70
80
90



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.

7.1.5 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. Let us import modules, inherit `torch.nn.Module` to 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 os
```

(续下页)

(接上页)

```
import time
import argparse
import numpy as np
from torch.cuda import amp
_seed_ = 2020
torch.manual_seed(_seed_)  # use torch.manual_seed() to seed the RNG for all devices␣
↪(both CPU and CUDA)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
np.random.seed(_seed_)

class PythonNet(nn.Module):
    def __init__(self, T):
        super().__init__()
        self.T = T
```

Then we add convolutional layers and a fully-connected layers to `PythonNet`. We add two Conv-BN-Pooling:

```
.. code-block:: python
```

self.conv = nn.Sequential(

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

The input with `shape=[N, 1, 28, 28]` will be converted to spikes with `shape=[N, 128, 7, 7]`.

Such convolutional layers can actually function as an encoder: in the previous tutorial (classify MNIST), we used a Poisson encoder to encode pictures into spikes. However, we can directly send the picture to the SNN. In this case, the first spike neurons layer (SN) and the layers before SN can be regarded as an auto-encoder with learnable parameters. Specifically, teh auto-encoder is composed of the following layers:

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

These layers receive images as input and output spikes, which can be regarded as an encoder.

Next, we add two fully-connected layers as the classifier. There are 10 neurons in output layer because the classes number in Fashion-MNIST is 10.

```
self.fc = nn.Sequential(
    nn.Flatten(),
```

(续下页)

(接上页)

```
    nn.Linear(128 * 7 * 7, 128 * 4 * 4, bias=False),
    neuron.IFNode(surrogate_function=surrogate.ATan()),
    nn.Linear(128 * 4 * 4, 10, bias=False),
    neuron.IFNode(surrogate_function=surrogate.ATan()),
)
```

Now let us define the forward function.

```
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
```

Avoid Duplicated Computing

We can train this network directly, just like the previous MNIST classification. But if we re-examine the structure of the network, we can find that some calculations are duplicated. For the first two layers of the network (the highlighted part of the following codes):

```
self.conv = nn.Sequential(
        nn.Conv2d(1, 128, kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(128),
        neuron.IFNode(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(surrogate_function=surrogate.ATan()),
        nn.MaxPool2d(2, 2)  # 7 * 7
    )
```

The input images are static and do not change with `t`. But they will be involved in `for` loop. At each time-step, they will flow through the first two layers with the same calculation. We can remove them from `for` loop in time-steps. The complete codes are:

```
class PythonNet(nn.Module):
    def __init__(self, T):
        super().__init__()
        self.T = T
```

(续下页)

(接上页)

```
        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(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(surrogate_function=surrogate.ATan()),
            nn.MaxPool2d(2, 2)  # 7 * 7

        )
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128 * 7 * 7, 128 * 4 * 4, bias=False),
            neuron.IFNode(surrogate_function=surrogate.ATan()),
            nn.Linear(128 * 4 * 4, 10, bias=False),
            neuron.IFNode(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
```

We put these stateless layers to `self.static_conv` to avoid duplicated calculations.

Training network

The complete codes are available at *spikingjelly.clock_driven.examples.conv_fashion_mnist*. The tarining arguments are:

```
Classify Fashion-MNIST

optional arguments:
  -h, --help            show this help message and exit
  -T T                  simulating time-steps
  -device DEVICE        device
  -b B                  batch size
  -epochs N             number of total epochs to run
  -j N                  number of data loading workers (default: 4)
  -data_dir DATA_DIR    root dir of Fashion-MNIST dataset
  -out_dir OUT_DIR      root dir for saving logs and checkpoint
  -resume RESUME        resume from the checkpoint path
  -amp                  automatic mixed precision training
  -cupy                 use cupy neuron and multi-step forward mode
  -opt OPT              use which optimizer. SDG or Adam
  -lr LR                learning rate
  -momentum MOMENTUM    momentum for SGD
  -lr_scheduler LR_SCHEDULER
                        use which schedule. StepLR or CosALR
  -step_size STEP_SIZE  step_size for StepLR
  -gamma GAMMA          gamma for StepLR
  -T_max T_MAX          T_max for CosineAnnealingLR
```

The checkpoint 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.

```
(pytorch-env) root@e8b6e4800dae4011eb0918702bd7ddedd51c-fangw1598-0:/# python -m␣
↪spikingjelly.clock_driven.examples.conv_fashion_mnist -opt SGD -data_dir /userhome/
↪datasets/FashionMNIST/ -amp

Namespace(T=4, T_max=64, amp=True, b=128, cupy=False, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
PythonNet(
  (static_conv): Sequential(
    (0): Conv2d(1, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
  )
  (conv): Sequential(
```

(续下页)

(接上页)

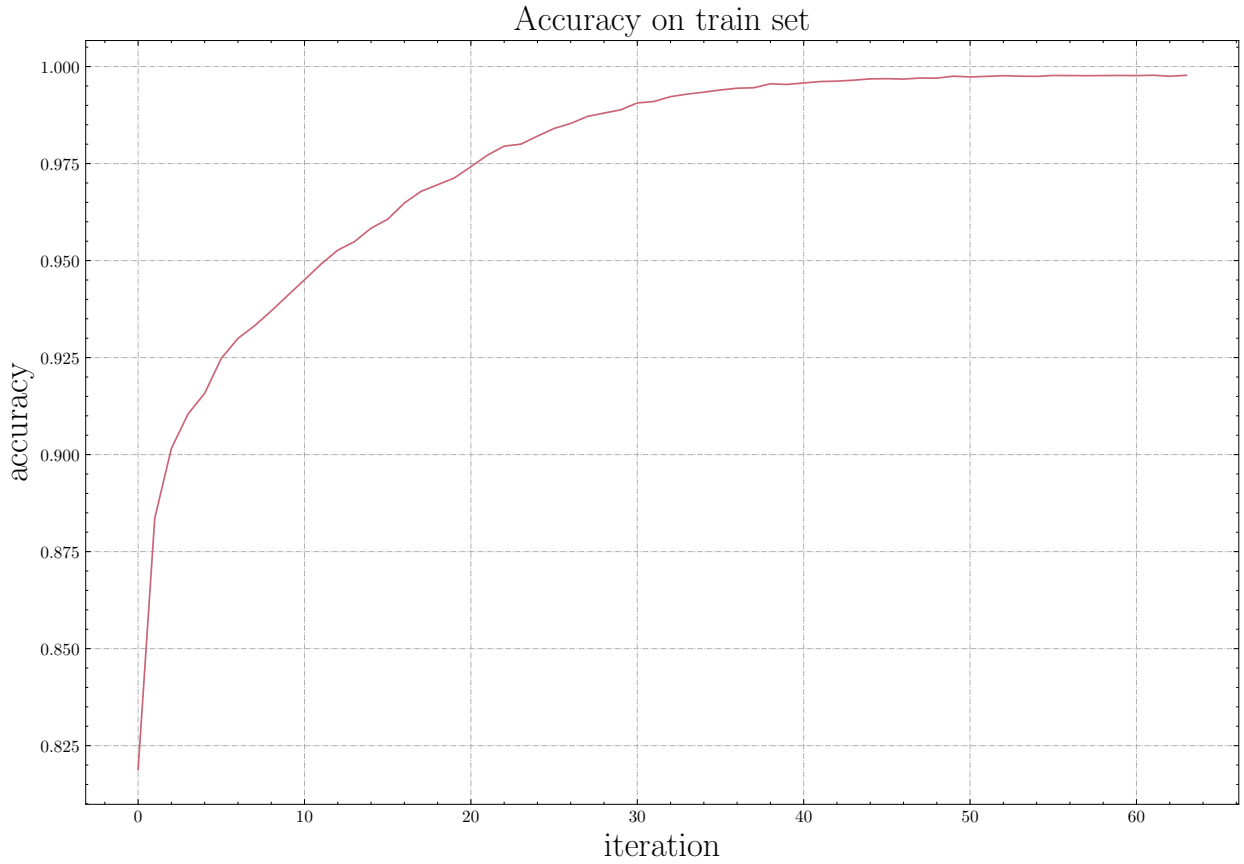
```
    (0): IFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
    (4): IFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=6272, out_features=2048, bias=False)
    (2): IFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (3): Linear(in_features=2048, out_features=10, bias=False)
    (4): IFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
  )
)
Mkdir ./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp.
Namespace(T=4, T_max=64, amp=True, b=128, cupy=False, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp
epoch=0, train_loss=0.028124165828697957, train_acc=0.8188267895299145, test_loss=0.
↪023525000348687174, test_acc=0.8633, max_test_acc=0.8633, total_time=16.
↪86261749267578
Namespace(T=4, T_max=64, amp=True, b=128, cupy=False, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp
epoch=1, train_loss=0.018544567498163536, train_acc=0.883613782051282, test_loss=0.
↪02161250041425228, test_acc=0.8745, max_test_acc=0.8745, total_time=16.
```

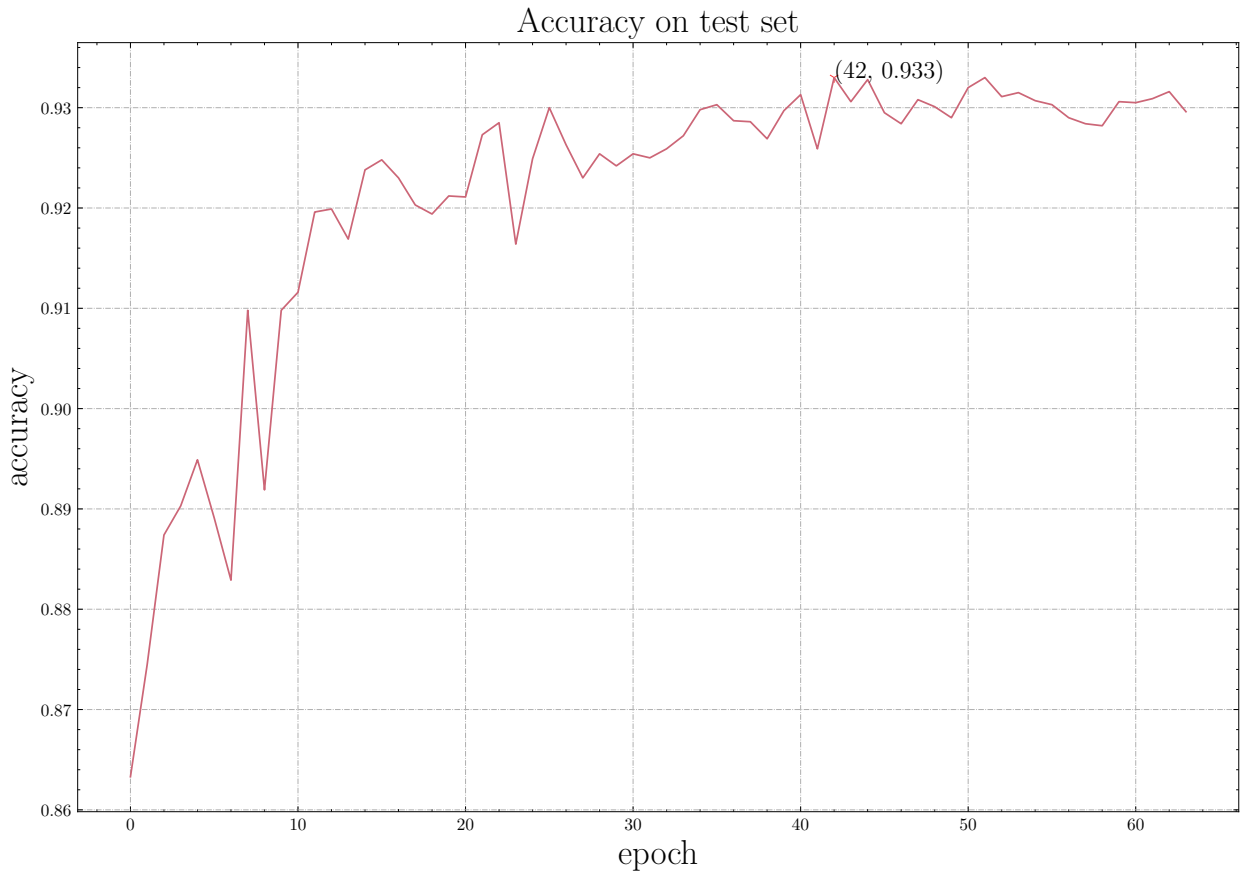
(续下页)

(接上页)

```
↪618073225021362
Namespace(T=4, T_max=64, amp=True, b=128, cupy=False, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)

...

./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp
epoch=62, train_loss=0.0010829827882937538, train_acc=0.997512686965812, test_loss=0.
↪011441250185668468, test_acc=0.9316, max_test_acc=0.933, total_time=15.
↪976636171340942
Namespace(T=4, T_max=64, amp=True, b=128, cupy=False, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp
epoch=63, train_loss=0.0010746361010835525, train_acc=0.9977463942307693, test_loss=0.
↪01154562517106533, test_acc=0.9296, max_test_acc=0.933, total_time=15.83976149559021
```

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





After training for 64 epochs, the highest test set accuracy rate can reach 93.3%, which is a very good accuracy for SNN. It is only slightly lower than ResNet18 (93.3%) with Normalization, random horizontal flip, random vertical flip, random translation and random rotation in the BenchMark Fashion-MNIST.

Visual Encoder

As we said in the above text, the first spike neurons layer (SN) and the layers before SN can be regarded as an auto-encoder with learnable parameters. Specifically, it is the highlighted part of our network shown below:

```
class Net(nn.Module):
    def __init__(self, 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(surrogate_function=surrogate.ATan()),
        ...
```

(续下页)

(接上页)

```
        )
```

Now let's take a look at the output spikes 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 pictures one by one:

```
from matplotlib import pyplot as plt
import numpy as np
from spikingjelly.clock_driven.examples.conv_fashion_mnist import PythonNet
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)
```

We load net from the checkpoint:

```
net = torch.load('./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp/checkpoint_max.pth', 'cpu
↪')['net']
encoder = nn.Sequential(
    net.static_conv,
    net.conv[0]
)
encoder.eval()
```

Let us extract a image from the data set, send it to the encoder, and check the accumulated value $\sum_t S_t$ of the output spikes. In order to show clearly, we also normalize the pixel values of the output `feature_map` with linearly transformation to `[0, 1]`.

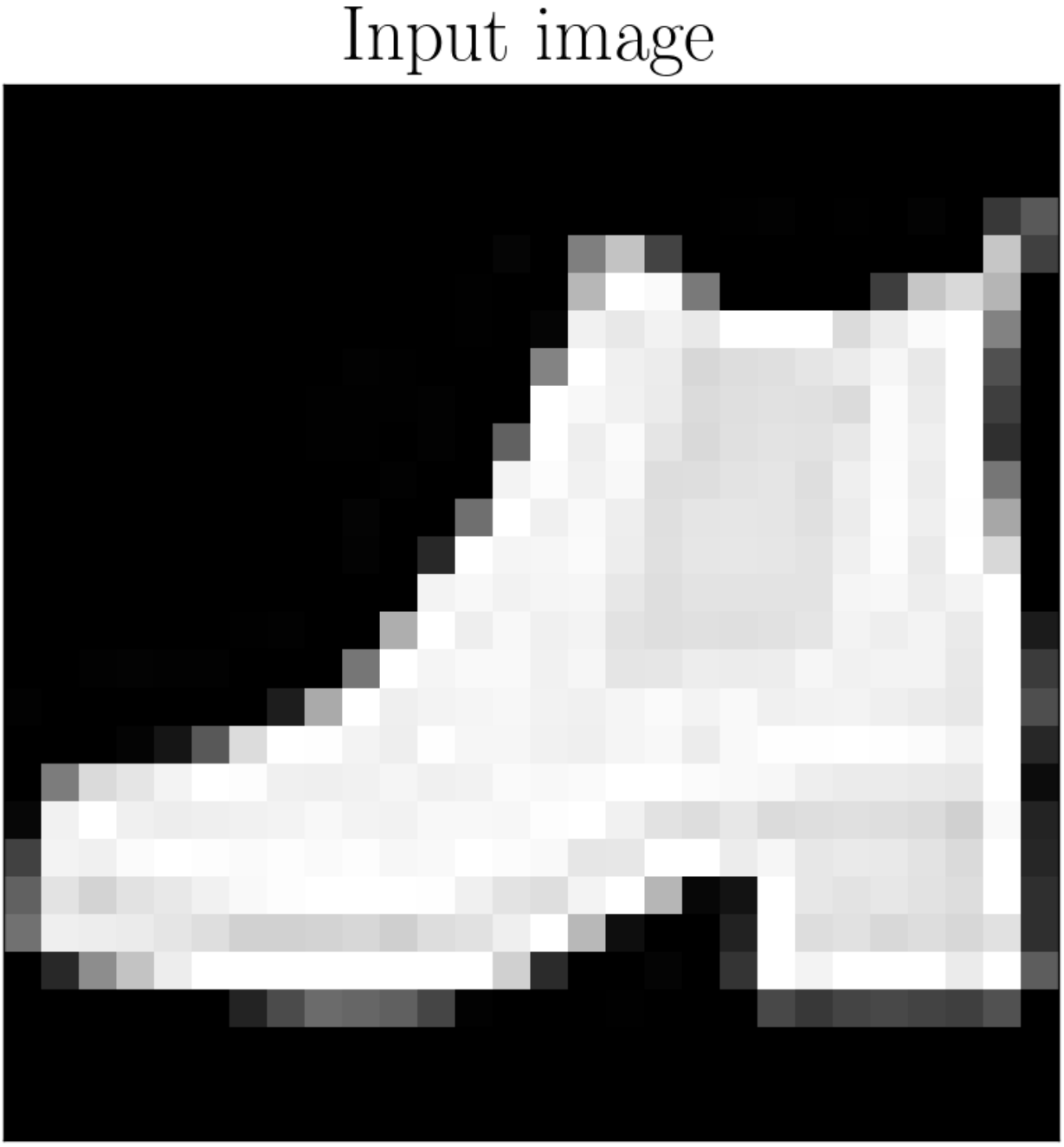
```
with torch.no_grad():
    # every time all the data sets are traversed, test once on the test set
    for img, label in test_data_loader:
        fig = plt.figure(dpi=200)
        plt.imshow(img.squeeze().numpy(), cmap='gray')
        # Note that the size of the image input to the network is ``[1,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

(续下页)

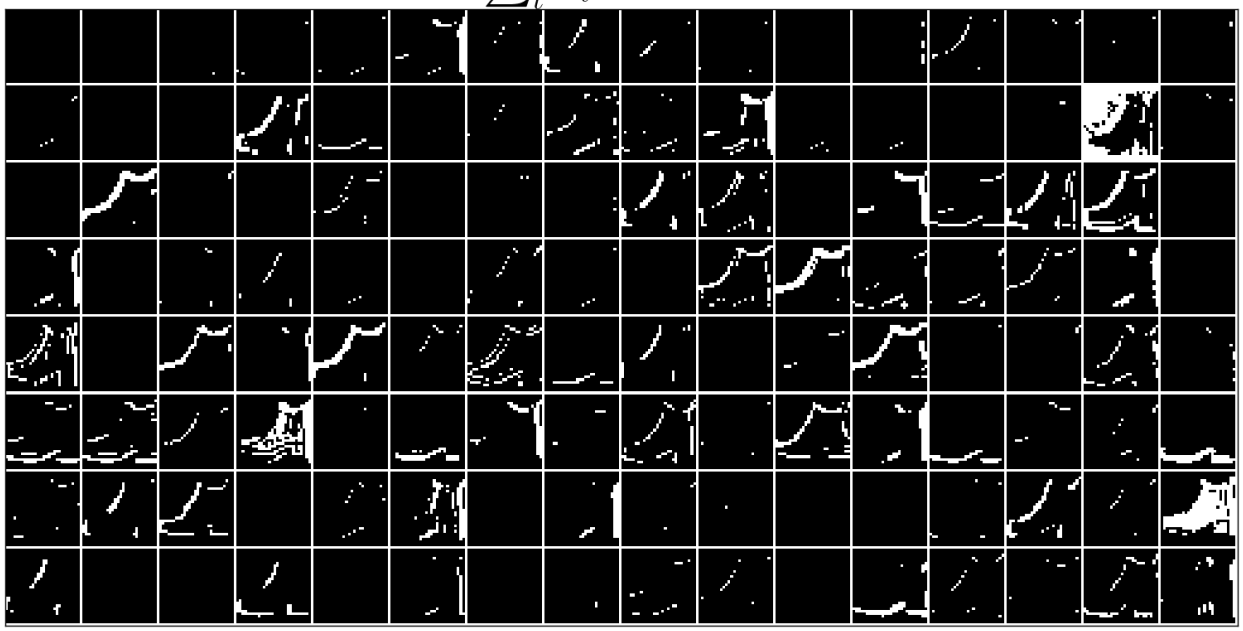
(接上页)

```
        # therefore, when calling ``imshow``, first use ``squeeze()`` to change the␣
↪size to ``[28, 28]``
        plt.title('Input image', fontsize=20)
        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()
```

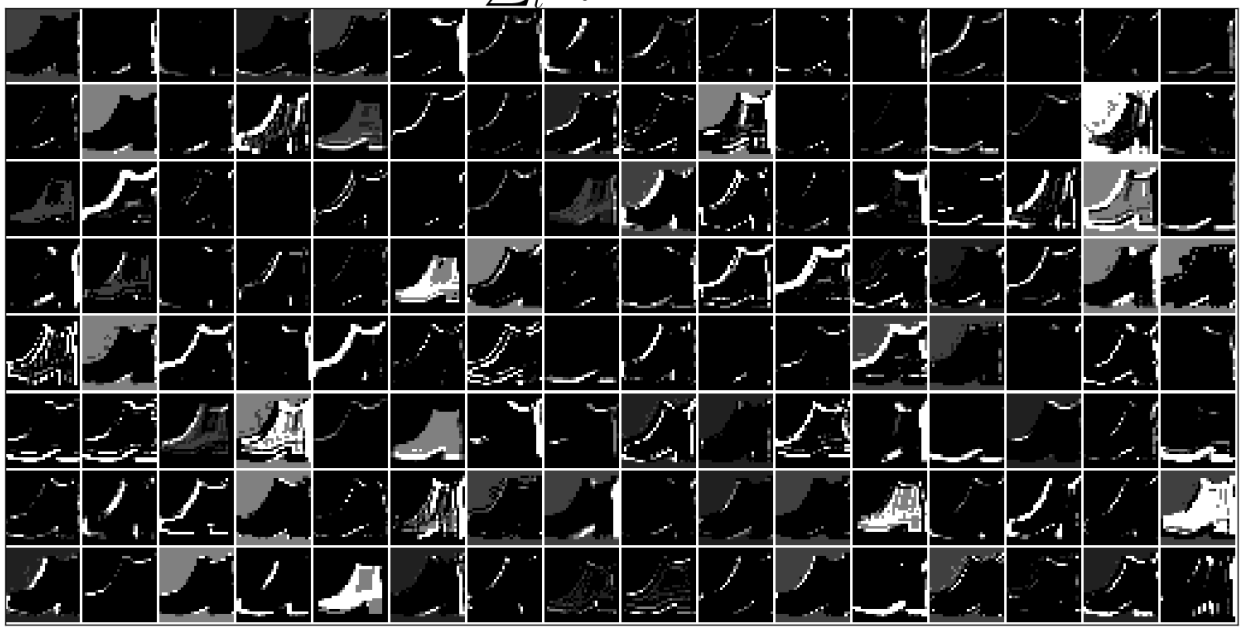
The following figure shows two input iamges and the cumulative spikes $\sum_t S_t$ encoded by the encoder at `t=0` and `t=7`:

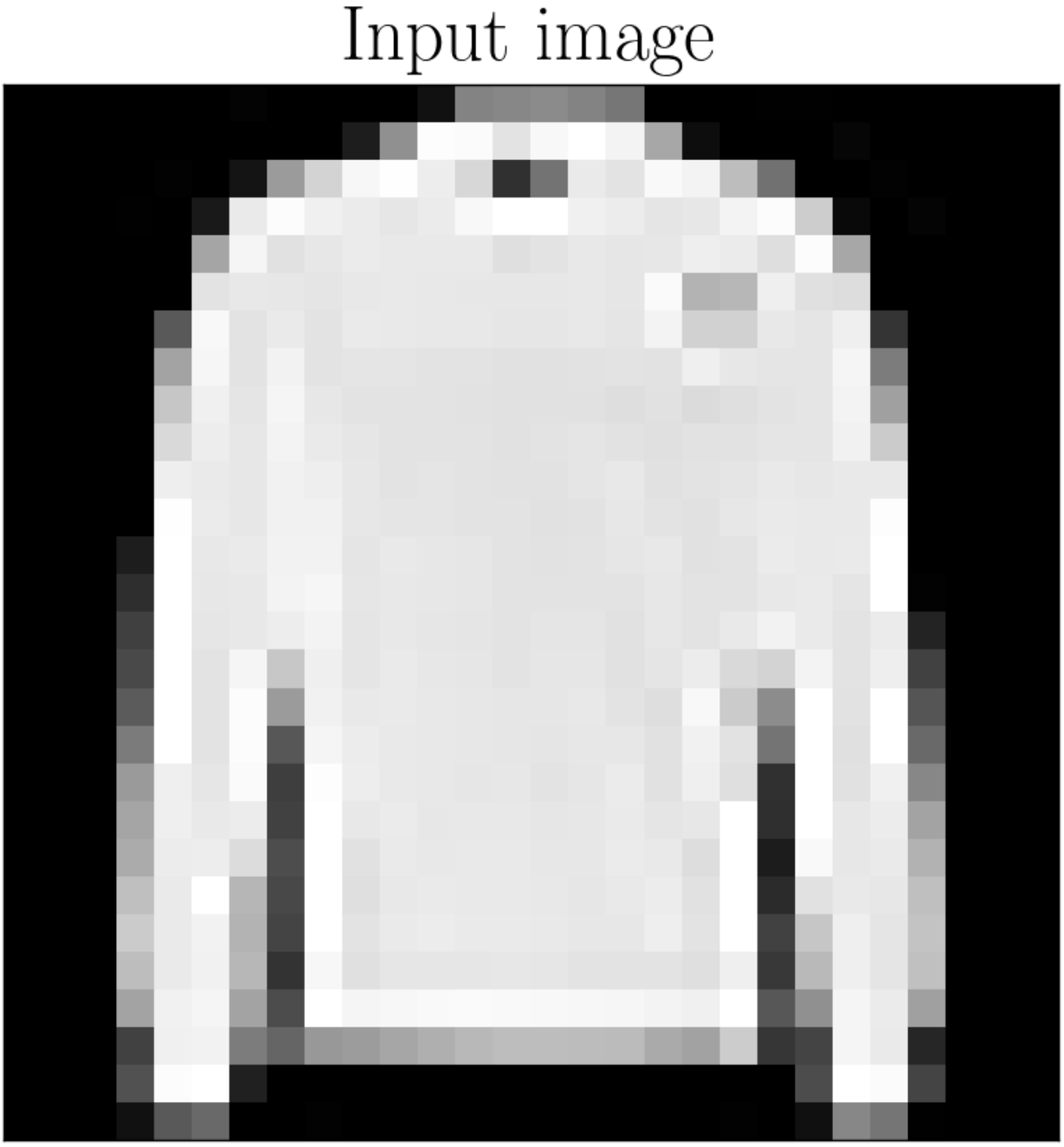

Input image

$\sum_t S_t$ at $t = 0$

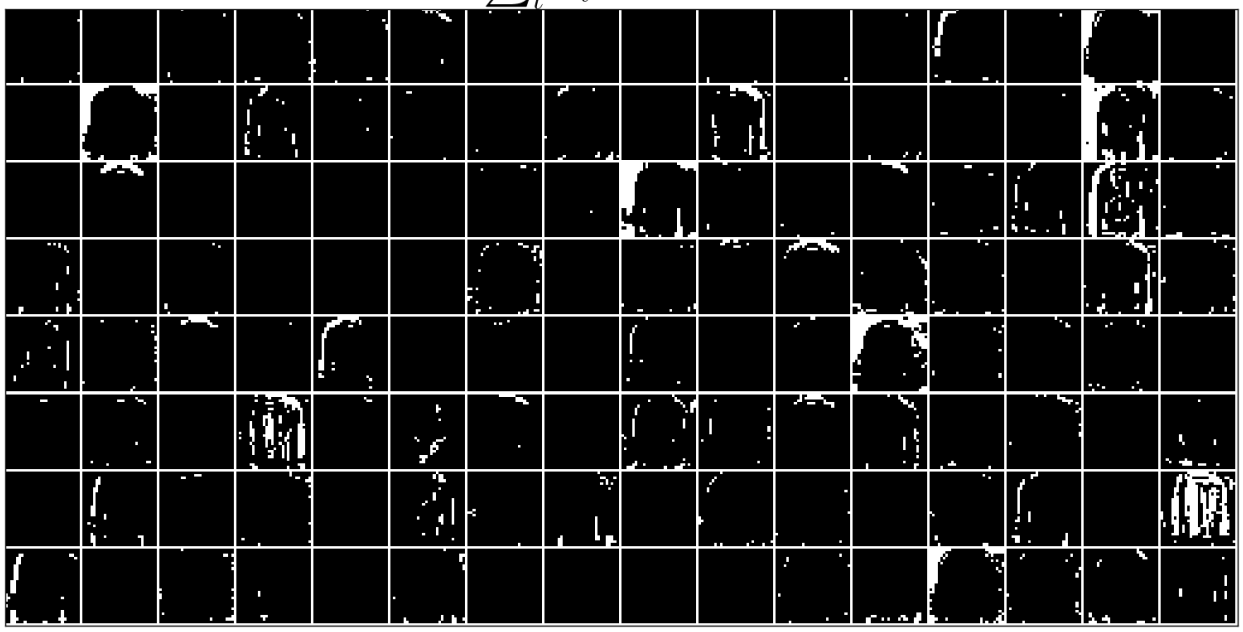


$\sum_t S_t$ at $t = 7$

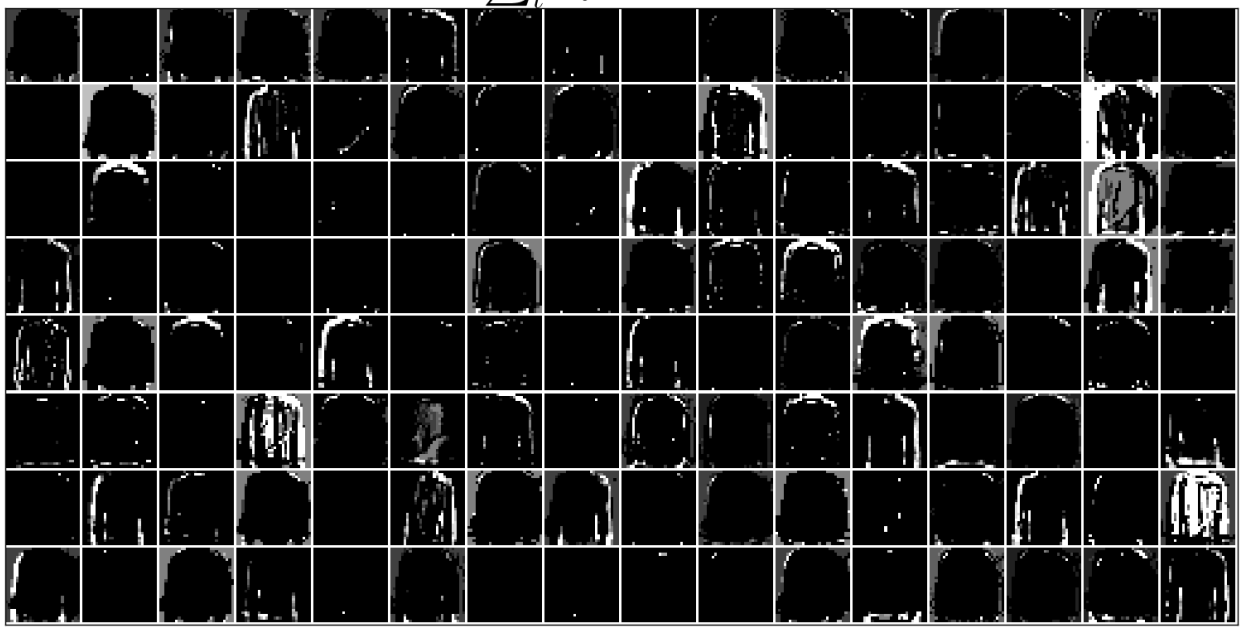



Input image

$\sum_t S_t$ at $t = 0$



$\sum_t S_t$ at $t = 7$



It can be found that the cumulative spikes $\sum_t S_t$ are very similar to the origin images, indicating that the encoder has strong coding ability.

7.1.6 spikingjelly.clock_driven.ann2snn

Author: DingJianhao, fangwei123456

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

There are two sets of implementations in earlier implementations: ONNX-based and PyTorch-based. Due to the instability of ONNX, this version is an enhanced version of PyTorch, which natively supports complex topologies (such as ResNet). Let’ s have a look!

Theoretical basis of ANN2SNN

Compared with ANN, the generated pulses of SNN are discrete, which is conducive to efficient communication. Today, with the popularity of ANN, the direct training of SNN requires more resources. Naturally, we will think of using the now very mature ANN to convert to SNN, and hope that SNN can have similar performance. This involves the problem of how to build a bridge between ANN and SNN. Now the mainstream way of SNN is to use frequency encoding, so for the output layer, we will use the number of neuron output pulses to judge the category. Is there a relationship between the release rate and ANN?

Fortunately, there is a strong correlation between the nonlinear activation of ReLU neurons in ANN and the firing rate of IF neurons in SNN (reset by subtracting the threshold: math:*V_{threshold}*). this feature to convert. The neuron update method mentioned here is the Soft method mentioned in Time-driven tutorial.

Experiment: Relationship between IF neuron spiking frequency and input

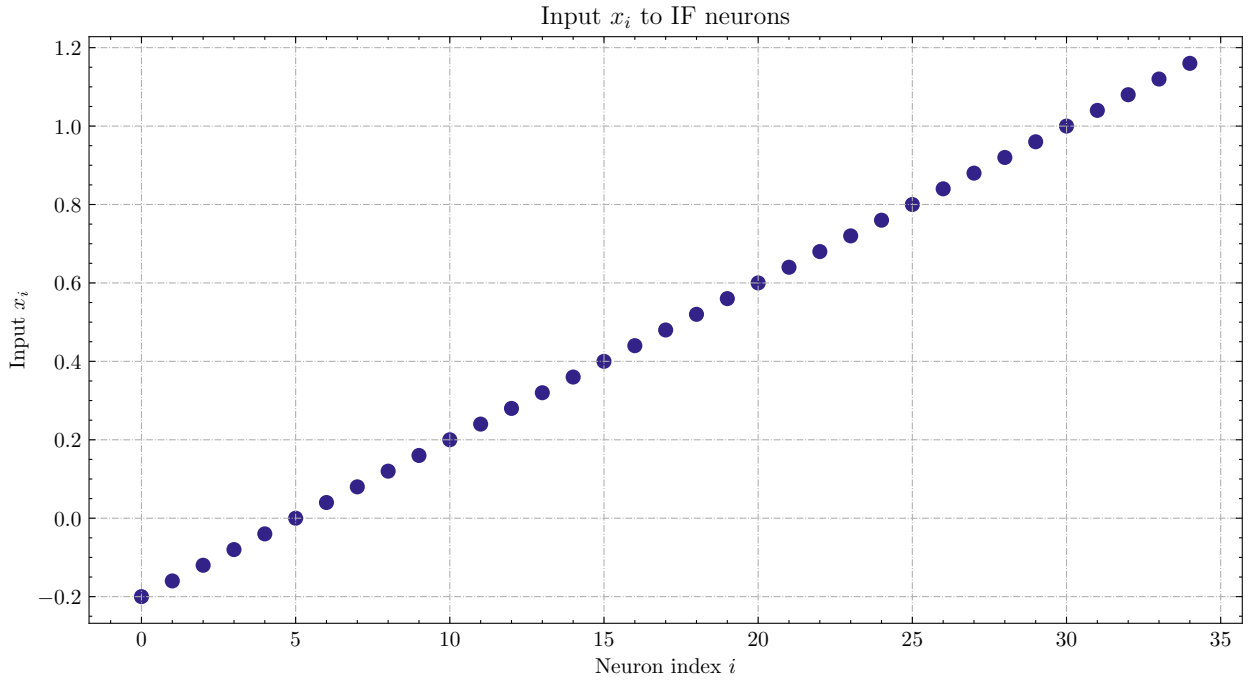
We gave constant input to the IF neuron and observed its output spikes and spike firing frequency. First import the relevant modules, create a new IF neuron layer, determine the input and draw the input of each IF neuron x_i:

```
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)
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 $x_{i}$')
```

(续下页)

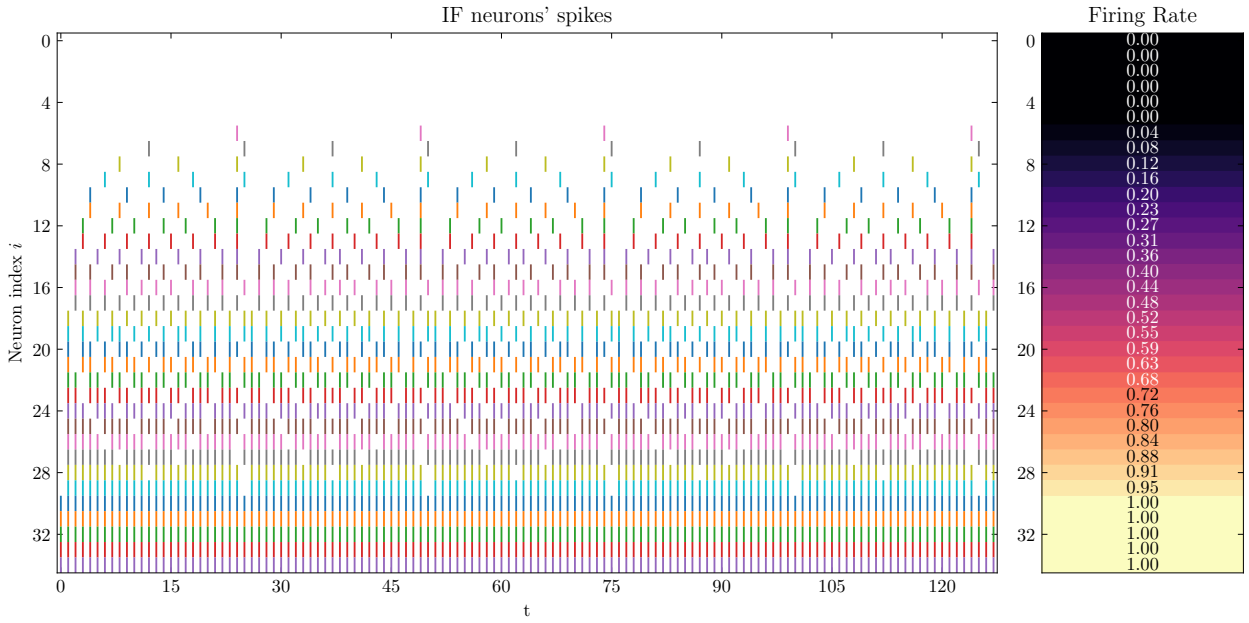
(接上页)

```
plt.grid(linestyle='-.')
plt.show()
```



Next, send the input to the IF neuron layer, and run the `T=128` step to observe the pulses and pulse firing frequency of each neuron:

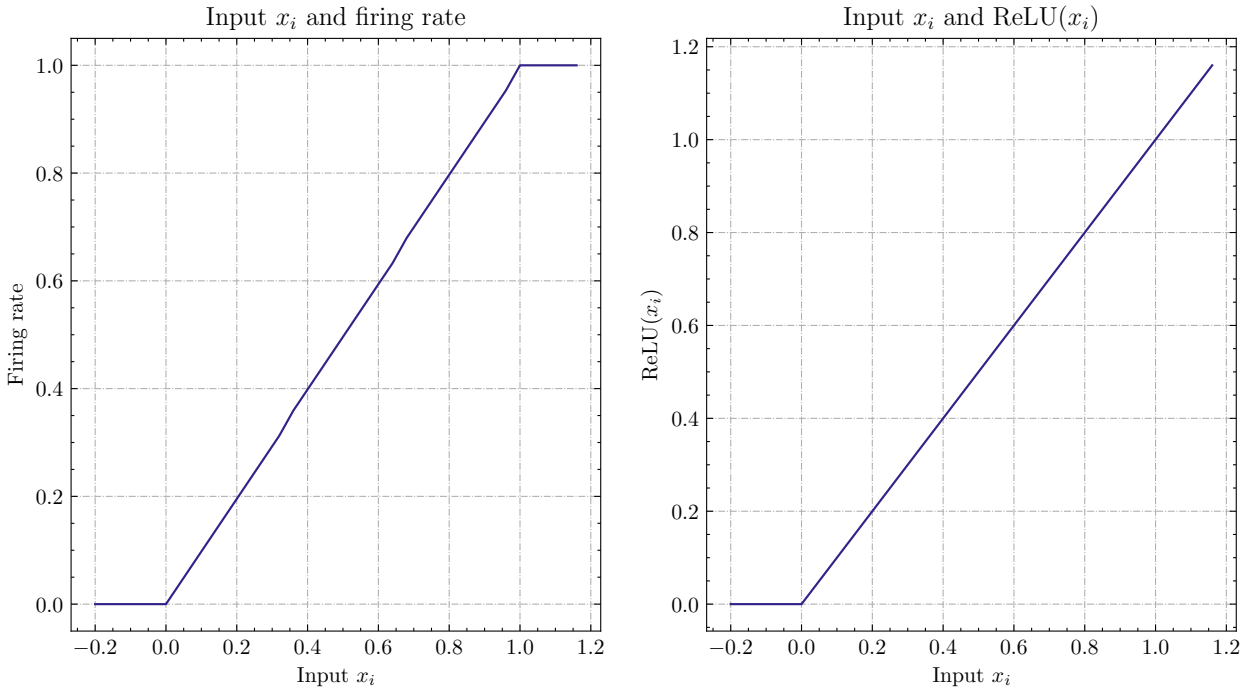
```
s_list = []
for t in range(T):
    s_list.append(if_node(x).unsqueeze(0))

out_spikes = np.asarray(torch.cat(s_list))
visualizing.plot_1d_spikes(out_spikes, 'IF neurons\' spikes and firing rates', 't',
↪'Neuron index $i$')
plt.show()
```



It can be found that the frequency of the pulse firing is within a certain range, which is proportional to the size of the input x_i.

Next, let' s plot the firing frequency of the IF neuron against the input x_i and compare it with ReLU(x_i):

```
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()
```



It can be found that the two curves are almost the same. It should be noted that the pulse frequency cannot be higher than 1, so the IF neuron cannot fit the input of the ReLU in the ANN is larger than 1.

Theoretical basis of ANN2SNN

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[Page 208, 1] 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[Page 208, 1]. The methods in ann2snn also mainly come from[Page 208, 1] .

Converting to spiking neural network

Conversion mainly solves two problems:

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

◆ BatchNorm parameter absorption

Assume that the parameters of BatchNorm are: math:*gamma* (`BatchNorm.weight`), β (`BatchNorm.bias`), μ (`BatchNorm. .running_mean`) , σ (`BatchNorm.running_var`, $\sigma = \sqrt{\text{running\_var}}$). For specific parameter definitions, see torch.nn.BatchNorm1d . Parameter modules (eg Linear) have parameters W and b . BatchNorm parameter absorption is to transfer the parameters of BatchNorm to W and b of the parameter module by operation, so that the output of the new module of data input is the same as when there is BatchNorm. For this, the $\bar{W}$ and $\bar{b}$ formulas for the new model are expressed as:

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

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

◆ Model Normalization

For a parameter module, it is assumed that its input tensor and output tensor are obtained, the maximum value of its input tensor is: math:*lambda_{pre}*, and the maximum value of its output tensor is: math:*lambda ‘Then, the normalized weight :math:*hat{W}‘ is:

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

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

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

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

So far, what we have done with neural networks is numerically equivalent. The current model should perform the same as the original model.

In the conversion, we need to change the ReLU activation function in the original model into IF neurons. For average pooling in ANN, we need to convert it to spatial downsampling. Since IF neurons can be equivalent to the ReLU activation function. Adding IF neurons or not after spatial downsampling has minimal effect on the results. There is currently no very ideal solution for max pooling in ANNs. The best solution so far is to control the pulse channel[Page 208, 1] with a gating function based on momentum accumulated pulses. Here we still recommend using avgpool2d. When simulating, according to the transformation theory, the SNN needs to input a constant analog input. Using a Poisson encoder will bring about a reduction in accuracy.

Implementation and optional configuration

The ann2snn framework will receive another major update in April 2022. The two categories of parser and simulator have been cancelled. Using the converter class replaces the previous solution. The current scheme is more compact and has more room for transformation settings.

◆ Converter class This class is used to convert ReLU’s ANN to SNN. Three common patterns are implemented here. The most common is the maximum current switching mode, which utilizes the upper and lower activation limits of the front and rear layers so that the case with the highest firing rate corresponds to the case where the activation achieves the maximum value. Using this mode requires setting the parameter mode to “max“[#f2]_. The 99.9% current switching mode utilizes the 99.9% activation quantile to limit the upper activation limit. Using this mode requires setting the parameter mode to “99.9%“[#f1]_. In the scaling conversion mode, the user needs to specify the scaling parameters into the mode, and the current can be limited by the activated maximum value after scaling. Using this mode requires setting the parameter mode to a float of 0-1.

Classify MNIST

Now we use `ann2snn` to build a simple convolutional network to classify the MNIST dataset.

First define our network structure (see `ann2snn.sample_models.mnist_cnn`):

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

(续下页)

(接上页)

```
        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)
        return x
```

Note: If you need to expand the tensor, define a `nn.Flatten` module in the network, and use the defined Flatten instead of the view function in the forward function.

Define our hyperparameters:

```
torch.random.manual_seed(0)
torch.cuda.manual_seed(0)
device = 'cuda'
dataset_dir = 'G:/Dataset/mnist'
batch_size = 100
T = 50
```

Here T is the inference time step used in inference for a while.

If you want to train, you also need to initialize the data loader, optimizer, loss function, for example:

Train the ANN. In the example, our model is trained for 10 epochs. The test set accuracy changes during training are as follows:

After training the model, we quickly load the model to test the performance of the saved model:

The output is as follows:

Converting with Converter is very simple, you only need to set the mode you want to use in the parameters. For example, to use MaxNorm, you need to define an `ann2snn.Converter` first, and forward the model to this object:

snn_model is the output SNN model.

Following this example, we define the modes as `max`, `99.9%`, `1.0/2`, `1.0/3`, `1.0/4`, `1.0/ 5` case SNN transformation and separate inference T steps to get the accuracy.

Observe the control bar output:

The speed of model conversion can be seen to be very fast. Model inference speed of 200 steps takes only 11s to complete (GTX 2080ti). Based on the time-varying accuracy of the model output, we can plot the accuracy for different settings.

Different settings can get different results, some inference speed is fast, but the final accuracy is low, and some inference is slow, but the accuracy is high. Users can choose model settings according to their needs.

7.1.7 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):
        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()
```

(续下页)

(接上页)

```
    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())
```

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)
```

(续下页)

(接上页)

```
        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())
```

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)
```

(续下页)

(接上页)

```
    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
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.

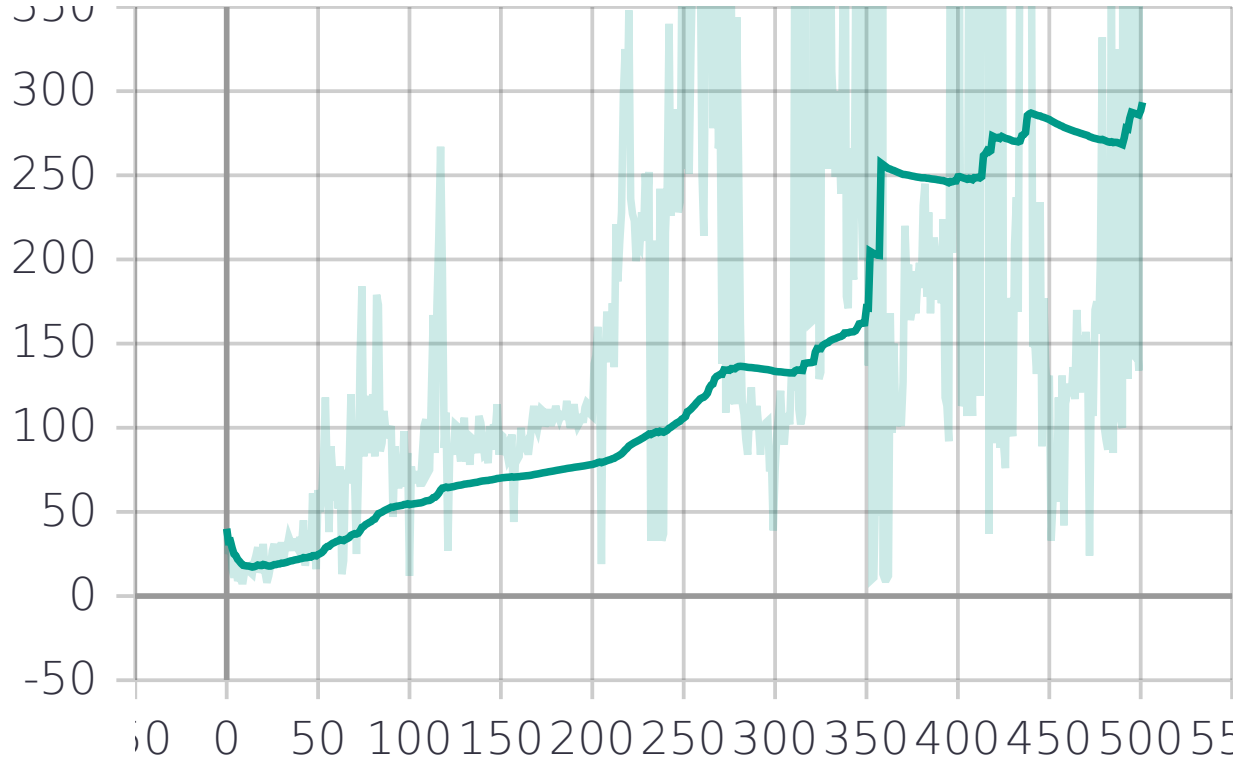
```
>>> from spikingjelly.clock_driven.examples import Spiking_DQN_state
>>> Spiking_DQN_state.play(use_cuda=False, pt_path='./model/CartPole-v0/policy_net_
↪256_max.pt', env_name='CartPole-v0', hidden_size=256, played_frames=300)
```

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.

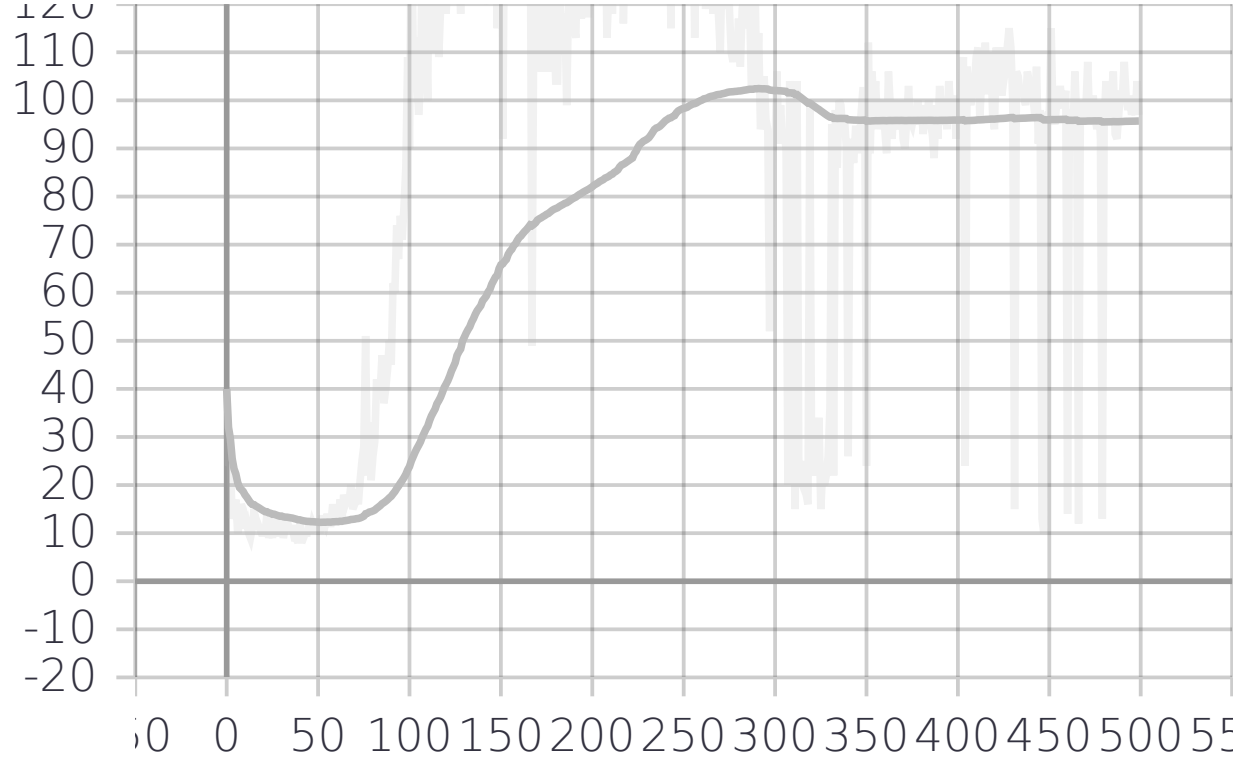
The performance after 16 episodes:

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).



7.1.8 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):
```

(续下页)

(接上页)

```
        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
```

(续下页)

(接上页)

```
        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()
```

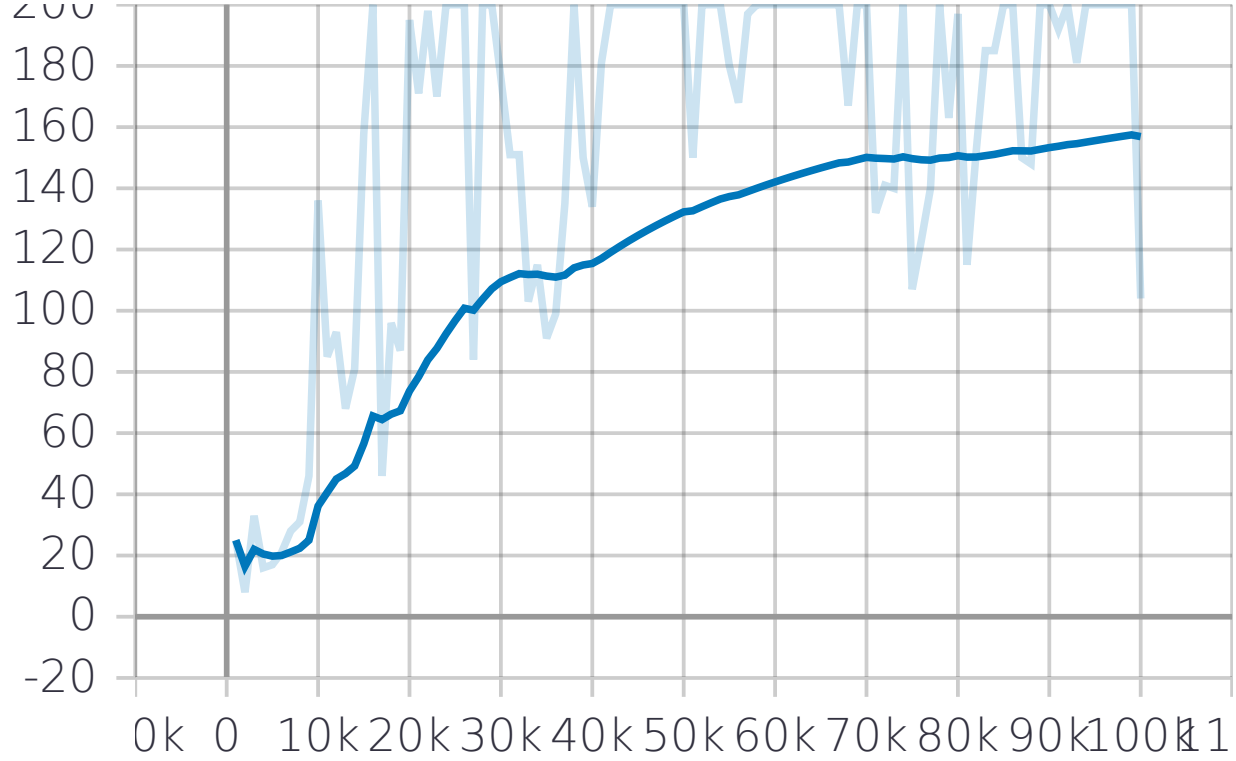
It should be emphasized here that, we need to `reset` the network after each forward process, because SNN is retentive while each trial should be started with a clean network state.

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.

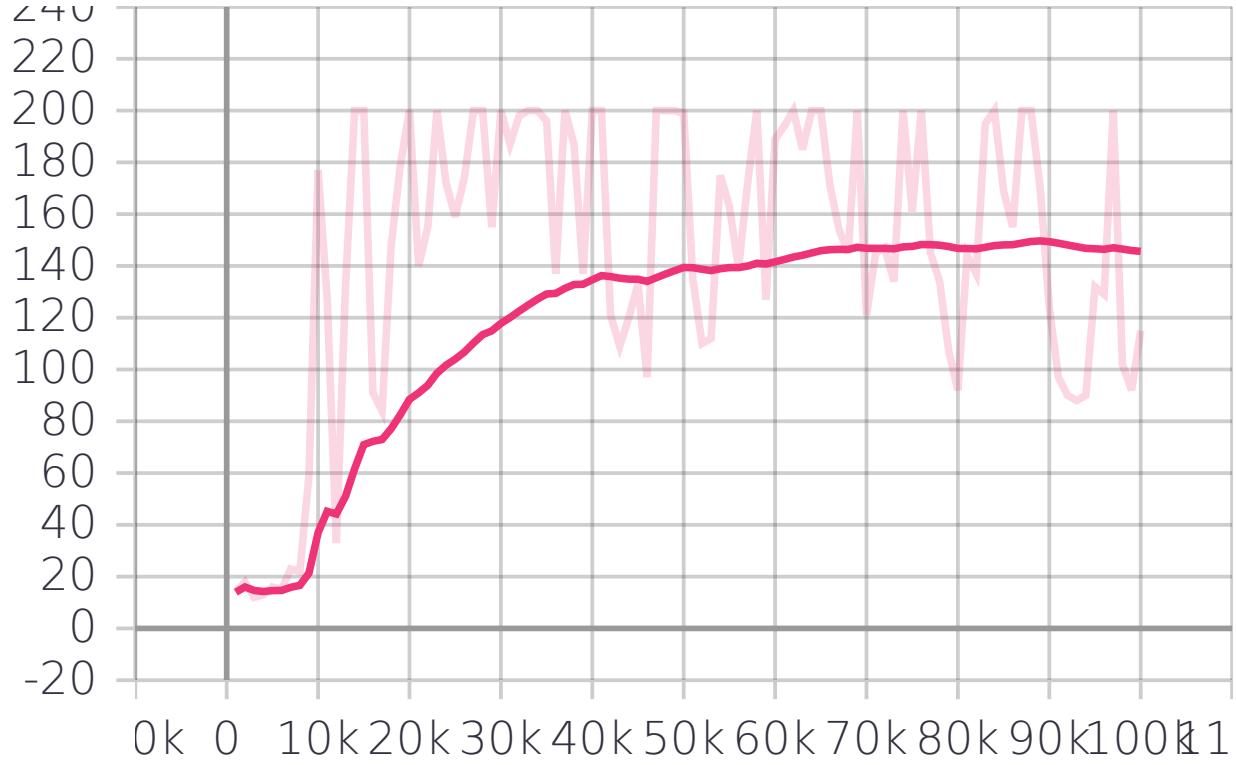
```
>>> python Spiking_A2C.py
```

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.



7.1.9 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
```

(续下页)

(接上页)

```
    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:
```

(续下页)

(接上页)

```
    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()
```

(续下页)

(接上页)

```
    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)
```

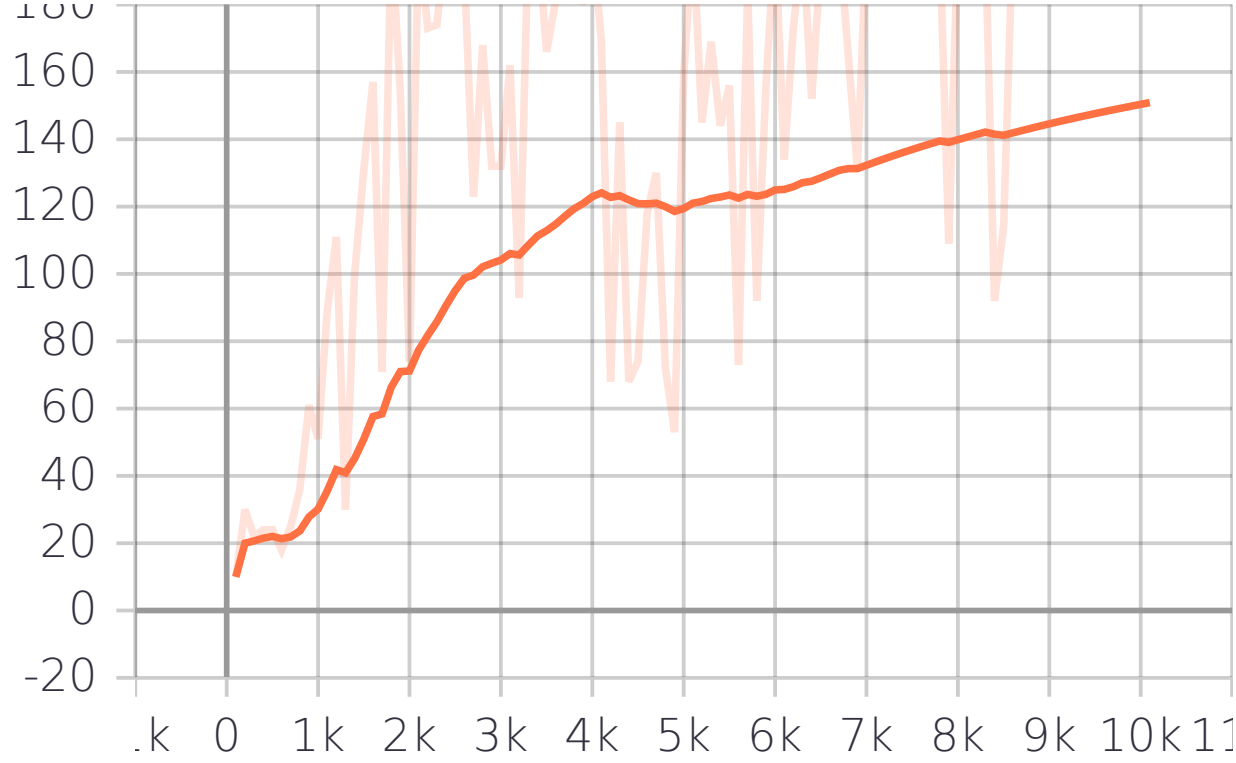
It should be emphasized here that, we need to `reset` the network after each forward process, because SNN is retentive while each trial should be started with a clean network state.

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.

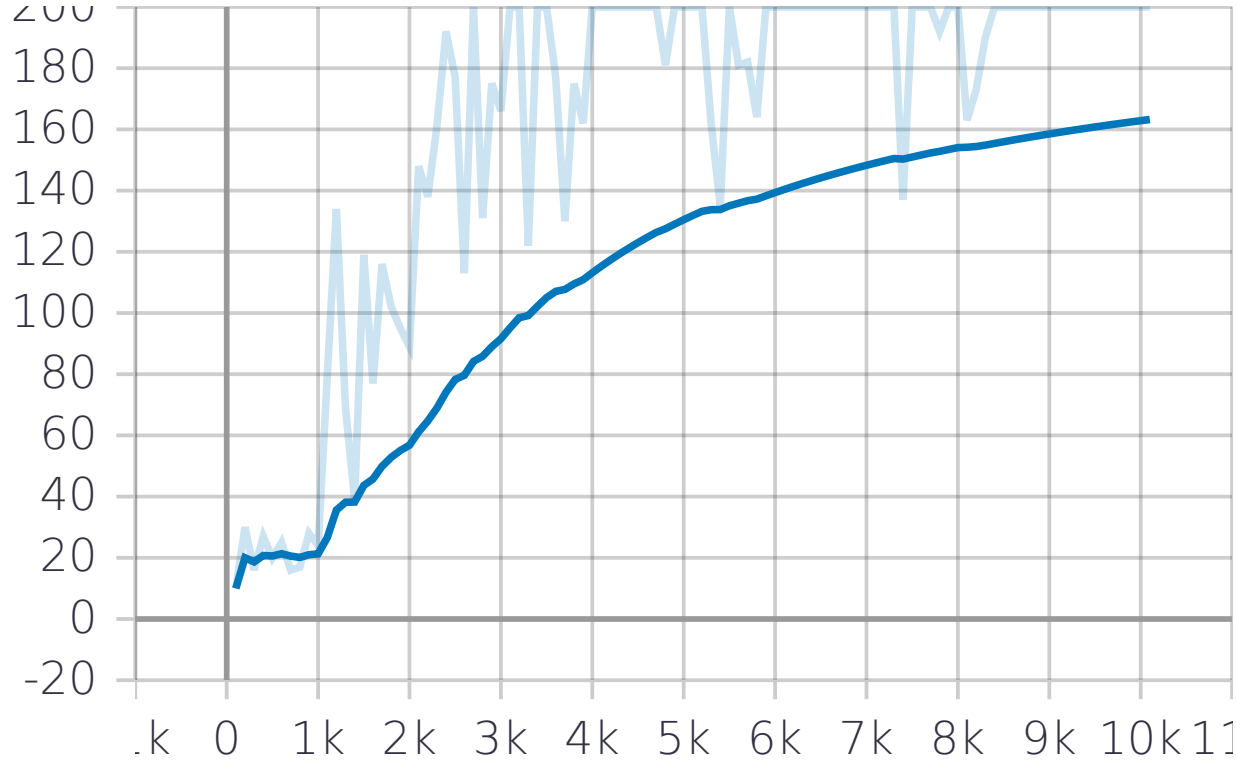
```
>>> python Spiking_PPO.py
```

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.



7.1.10 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 = {}
```

(续下页)

(接上页)

```
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
```

(续下页)

(接上页)

```
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 = []
```

(续下页)

(接上页)

```
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' % (
```

(续下页)

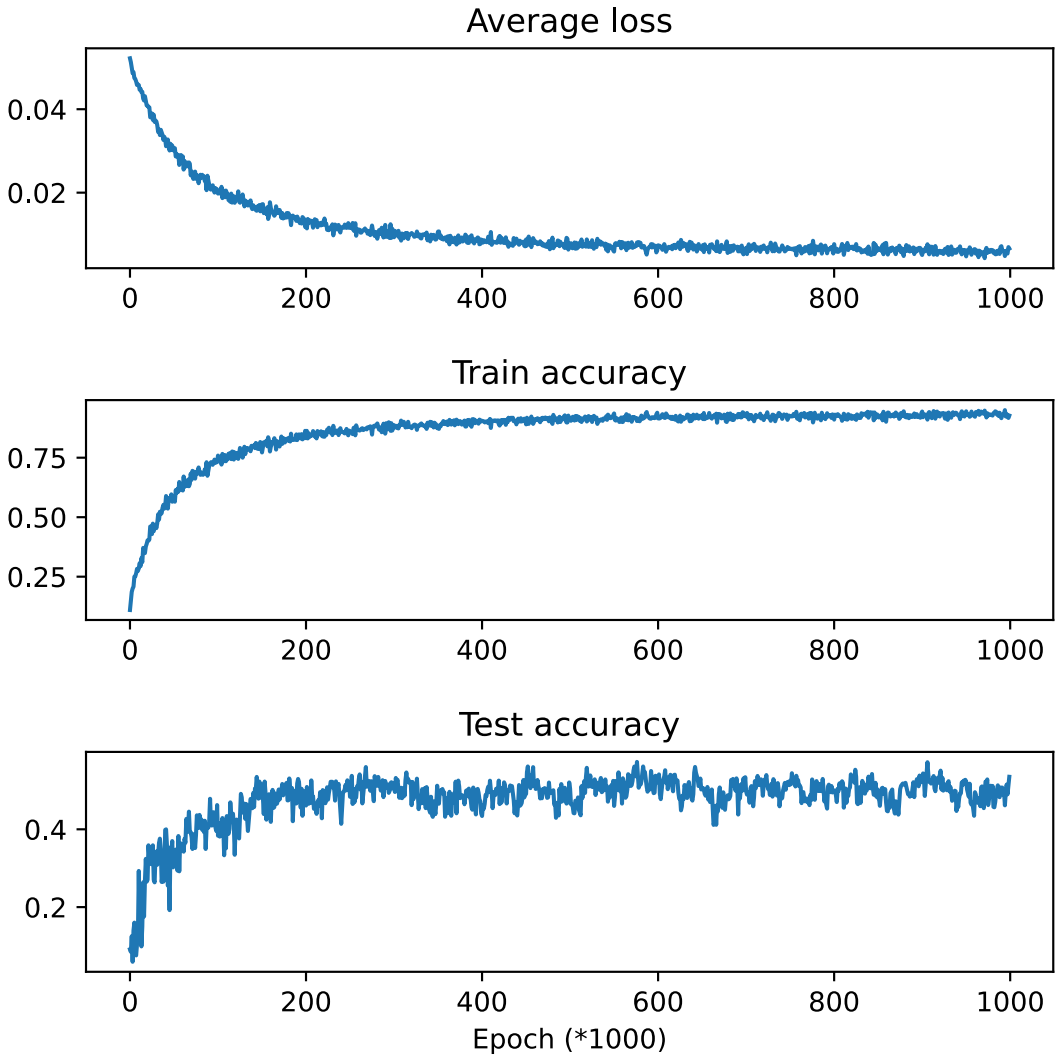
(接上页)

```
                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...')
```

(续下页)

(接上页)

```
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)
```

(续下页)

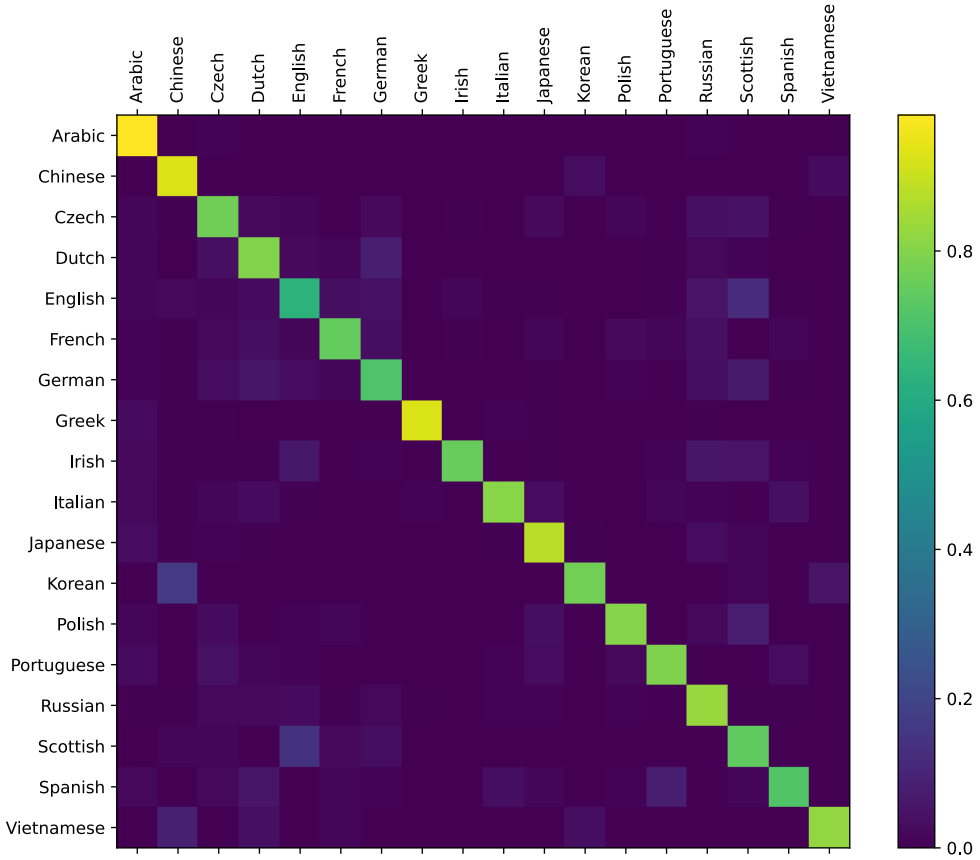
(接上页)

```
    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.



7.1.11 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.Sequential):
    def __init__(self, *args):
        super().__init__(*args)

    def forward(self, x_seq: torch.Tensor):
        """
        :param x_seq: shape=[T, batch_size, ...]
        :type x_seq: torch.Tensor
        :return: y_seq, shape=[T, batch_size, ...]
        :rtype: torch.Tensor
        """
        y_seq = []
        for t in range(x_seq.shape[0]):
            y_seq.append(super().forward(x_seq[t]))

        for t in range(y_seq.__len__()):
            y_seq[t] = y_seq[t].unsqueeze(0)
        return torch.cat(y_seq, 0)
```

Let us use *spikingjelly.clock_driven.layer.MultiStepContainer* to implement a multi-step IF neuron:

```
from spikingjelly.clock_driven import neuron, layer, functional
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]))
functional.reset_net(if_node)

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

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.])
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

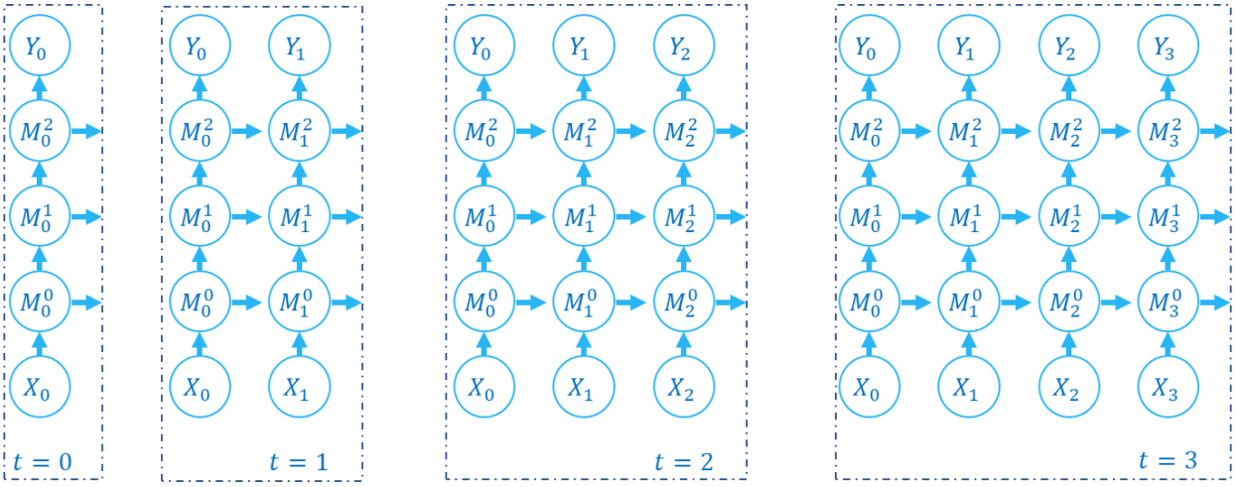
In the previous tutorials and examples, we run the SNNs *step-by-step*, e.g.,:

```
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])
```

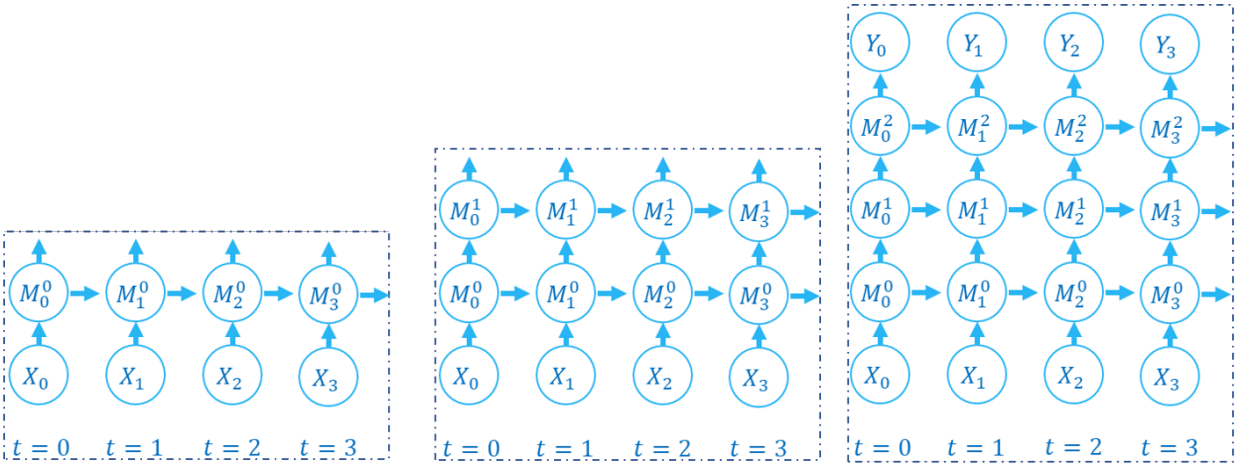
The computation graph of forward propagation is built as followed:



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.

Wrap Forward Propagation

After we use `SeqToANNContainer` to wrap stateless ANN's layers, the `.keys()` of network's `state_dict` will change because we introduce an external wrapper. Here is an example:

```
net_step_by_step = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=3, padding=1, bias=False),
    nn.BatchNorm2d(16),
    neuron.IFNode()
)

net_layer_by_layer = nn.Sequential(
    layer.SeqToANNContainer(
        nn.Conv2d(3, 16, kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(16),
    ),
    neuron.MultiStepIFNode()
)
```

(续下页)

(接上页)

```
print('net_step_by_step.state_dict:', net_step_by_step.state_dict().keys())
print('net_layer_by_layer.state_dict:', net_layer_by_layer.state_dict().keys())
```

The outputs are:

```
net_step_by_step.state_dict: odict_keys(['0.weight', '1.weight', '1.bias', '1.running_
↪mean', '1.running_var', '1.num_batches_tracked'])
net_layer_by_layer.state_dict: odict_keys(['0.0.weight', '0.1.weight', '0.1.bias', '0.
↪1.running_mean', '0.1.running_var', '0.1.num_batches_tracked'])
```

We can find that keys have been changed, which causes some trouble to load model' s weights. For example, if we want to build a multi-step Spiking ResNet-18 (*spikingjelly.clock_driven.model.spiking_resnet.spiking_resnet18*), and we want to load the pre-train model' s weights from ANN. If the network is built by `SeqToANNContainer`, it wil be not able to load weights from ANN because keys of `state_dict` are different. To avoid such problems, we can wrap forward propagation, rather than wrap layers. Here is an example:

```
class NetStepByStep(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(3, 16, kernel_size=3, padding=1, bias=False)
        self.bn = nn.BatchNorm2d(16)
        self.sn = neuron.IFNode()

    def forward(self, x):
        # x.shape = [N, C, H, W]
        x = self.conv(x)
        x = self.bn(x)
        x = self.sn(x)
        return x


class NetLayerByLayer1(NetStepByStep):

    def forward(self, x_seq):
        # x_seq.shape = [T, N, C, H, W]
        x_seq = functional.seq_to_ann_forward(x_seq, [self.conv, self.bn])
        x_seq = functional.multi_step_forward(x_seq, self.sn)
        return x_seq


class NetLayerByLayer2(NetStepByStep):
    def __init__(self):
        super().__init__()
```

(续下页)

(接上页)

```
        # replace single-step neuron to multi-step neuron
        del self.sn
        self.sn = neuron.MultiStepIFNode()

    def forward(self, x_seq):
        # x_seq.shape = [T, N, C, H, W]
        x_seq = functional.seq_to_ann_forward(x_seq, [self.conv, self.bn])
        x_seq = self.sn(x_seq)
        return x_seq
```

`state_dict.keys()` of `NetStepByStep`, `NetLayerByLayer1`, `NetLayerByLayer2` are identical, and they can load model weights from each others.

7.1.12 Accelerate with CUDA-Enhanced Neuron and Layer-by-Layer Propagation

Authors: fangwei123456

CUDA-Enhanced Neuron

spikingjelly.clock_driven.neuron provides the multi-step version of neurons. Compared with the single-step neuron, the multi-step neuron can use cupy backend. The cupy backend fuses operations in a single cuda kernel, which is much faster than naive pytorch backend. Let us run a simple experiment to compare LIF neurons in both module:

```
from spikingjelly.clock_driven import neuron, surrogate, cu_kernel_opt
import torch


def cal_forward_t(multi_step_neuron, x, repeat_times):
    with torch.no_grad():
        used_t = cu_kernel_opt.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):
```

(续下页)

(接上页)

```
    x.requires_grad_(True)
    used_t = cu_kernel_opt.cal_fun_t(repeat_times, x.device, forward_backward, multi_
↪step_neuron, x)
    return used_t * 1000


device = 'cuda:0'
repeat_times = 1024
ms_lif = neuron.MultiStepLIFNode(surrogate_function=surrogate.ATan(alpha=2.0))


ms_lif.to(device)
N = 2 ** 20
print('forward')
ms_lif.eval()
for T in [8, 16, 32, 64, 128]:
    x = torch.rand(T, N, device=device)
    ms_lif.backend = 'torch'
    print(T, cal_forward_t(ms_lif, x, repeat_times), end=', ')
    ms_lif.backend = 'cupy'
    print(cal_forward_t(ms_lif, x, repeat_times))

print('forward and backward')
ms_lif.train()
for T in [8, 16, 32, 64, 128]:
    x = torch.rand(T, N, device=device)
    ms_lif.backend = 'torch'
    print(T, cal_forward_backward_t(ms_lif, x, repeat_times), end=', ')
    ms_lif.backend = 'cupy'
    print(cal_forward_backward_t(ms_lif, x, repeat_times))
```

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:

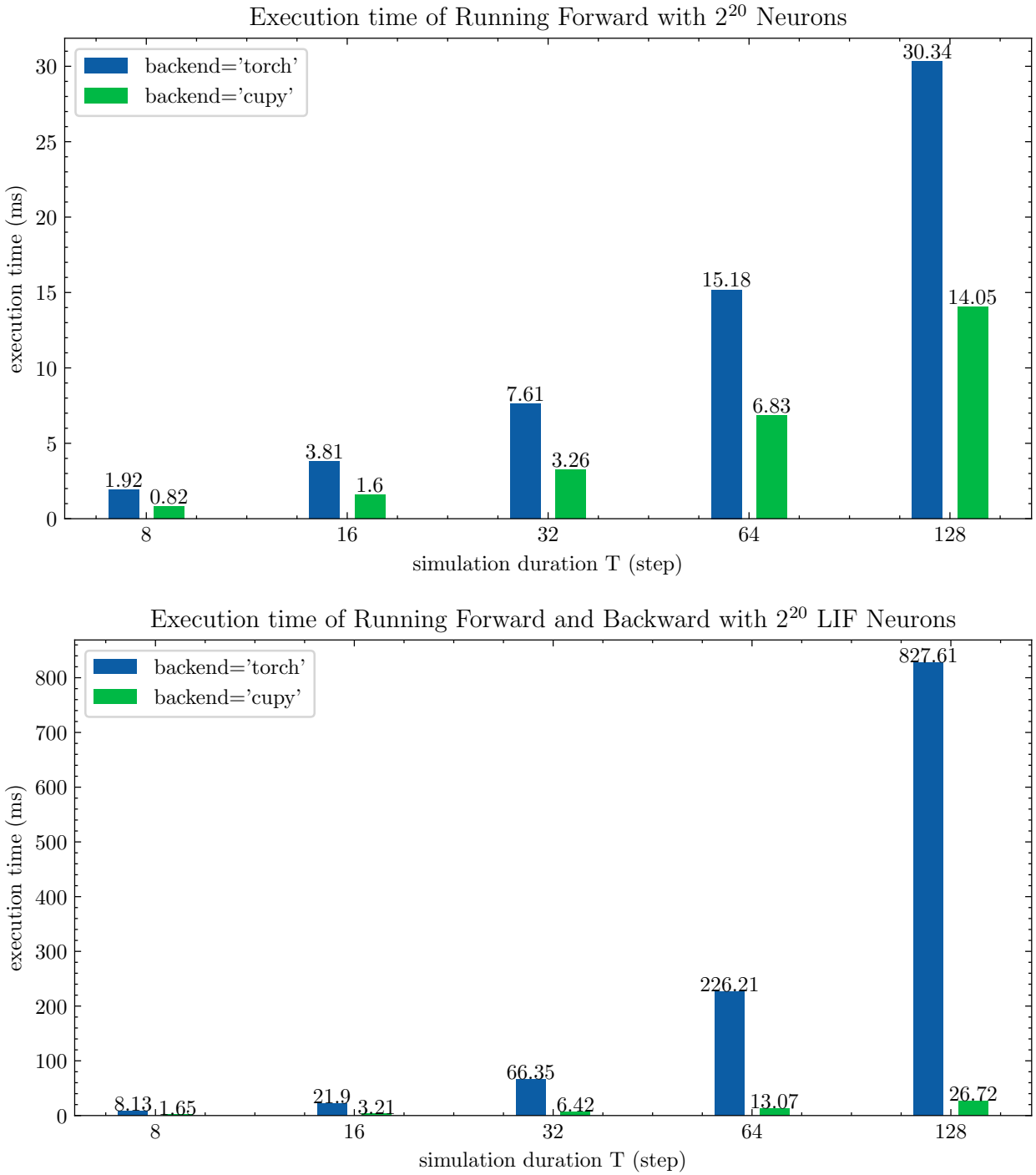
```
forward
8 1.9180845527841939, 0.8166529733273364
16 3.8143536958727964, 1.6002442711169351
32 7.6071328955436, 3.2570467449772877
64 15.181676714490777, 6.82808195671214
128 30.344632044631226, 14.053565065751172
forward and backward
8 8.131792200288146, 1.6501817200662572
16 21.89934094545265, 3.210343387223702
```

(续下页)

(接上页)

```
32 66.34630815216269, 6.41730432241161
64 226.20835550819152, 13.073845567419085
128 827.6064751953811, 26.71502177403795
```

We plot the results in a bar chart:



It can be found that cupy backend is much faster than naive pytorch backend.

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 CupyNet(nn.Module):
    def __init__(self, T):
        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.MultiStepIFNode(surrogate_function=surrogate.ATan(), backend='cupy
↪'),
            layer.SeqToANNContainer(
                    nn.MaxPool2d(2, 2),  # 14 * 14
                    nn.Conv2d(128, 128, kernel_size=3, padding=1, bias=False),
                    nn.BatchNorm2d(128),
            ),
            neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), backend='cupy
↪'),
            layer.SeqToANNContainer(
                nn.MaxPool2d(2, 2),  # 7 * 7
                nn.Flatten(),
            ),
        )
        self.fc = nn.Sequential(
            layer.SeqToANNContainer(nn.Linear(128 * 7 * 7, 128 * 4 * 4, bias=False)),
            neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), backend='cupy
↪'),
            layer.SeqToANNContainer(nn.Linear(128 * 4 * 4, 10, bias=False)),
            neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), backend='cupy
↪'),
        )


    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]
```

(续下页)

(接上页)

```
        return self.fc(self.conv(x_seq)).mean(0)
```

The fully codes are available at *spikingjelly.clock_driven.examples.conv_fashion_mnist*. Run this example with the same arguments and devices as those in *Clock driven: Use convolutional SNN to identify Fashion-MNIST*. The outputs are:

```
(pytorch-env) root@e8b6e4800dae4011eb0918702bd7ddedd51c-fangw1598-0:/# python -m␣
↪spikingjelly.clock_driven.examples.conv_fashion_mnist -opt SGD -data_dir /userhome/
↪datasets/FashionMNIST/ -amp -cupy

Namespace(T=4, T_max=64, amp=True, b=128, cupy=True, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
CupyNet(
  (static_conv): Sequential(
    (0): Conv2d(1, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
  )
  (conv): Sequential(
    (0): MultiStepIFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (1): SeqToANNContainer(
      (module): Sequential(
        (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
        (1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
        (2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (2): MultiStepIFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (3): SeqToANNContainer(
      (module): Sequential(
        (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
        (1): Flatten(start_dim=1, end_dim=-1)
```

(续下页)

(接上页)

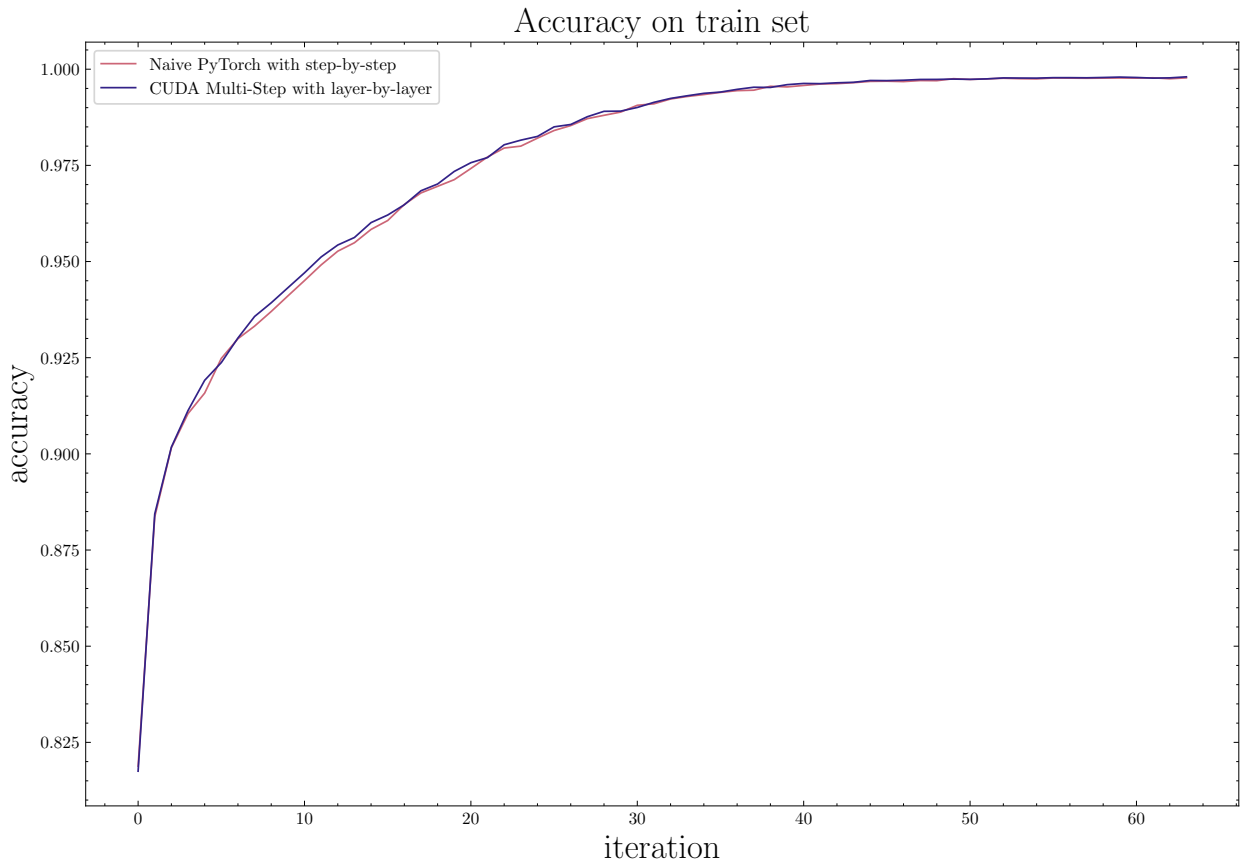
```
      )
    )
  )
  (fc): Sequential(
    (0): SeqToANNContainer(
      (module): Linear(in_features=6272, out_features=2048, bias=False)
    )
    (1): MultiStepIFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (2): SeqToANNContainer(
      (module): Linear(in_features=2048, out_features=10, bias=False)
    )
    (3): MultiStepIFNode(
      v_threshold=1.0, v_reset=0.0, detach_reset=False
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
  )
)
Mkdir ./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp_cupy.
Namespace(T=4, T_max=64, amp=True, b=128, cupy=True, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp_cupy
epoch=0, train_loss=0.028574782584865507, train_acc=0.8175080128205128, test_loss=0.
↪020883125430345536, test_acc=0.8725, max_test_acc=0.8725, total_time=13.
↪037598133087158
Namespace(T=4, T_max=64, amp=True, b=128, cupy=True, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp_cupy

...

epoch=62, train_loss=0.001055751721853287, train_acc=0.9977463942307693, test_loss=0.
↪010815625159442425, test_acc=0.934, max_test_acc=0.9346, total_time=11.
↪059867858886719
Namespace(T=4, T_max=64, amp=True, b=128, cupy=True, data_dir='/userhome/datasets/
↪FashionMNIST/', device='cuda:0', epochs=64, gamma=0.1, j=4, lr=0.1, lr_scheduler=
↪'CosALR', momentum=0.9, opt='SGD', out_dir='./logs', resume=None, step_size=32)
./logs/T_4_b_128_SGD_lr_0.1_CosALR_64_amp_cupy
epoch=63, train_loss=0.0010632637413514631, train_acc=0.9980134882478633, test_loss=0.
```

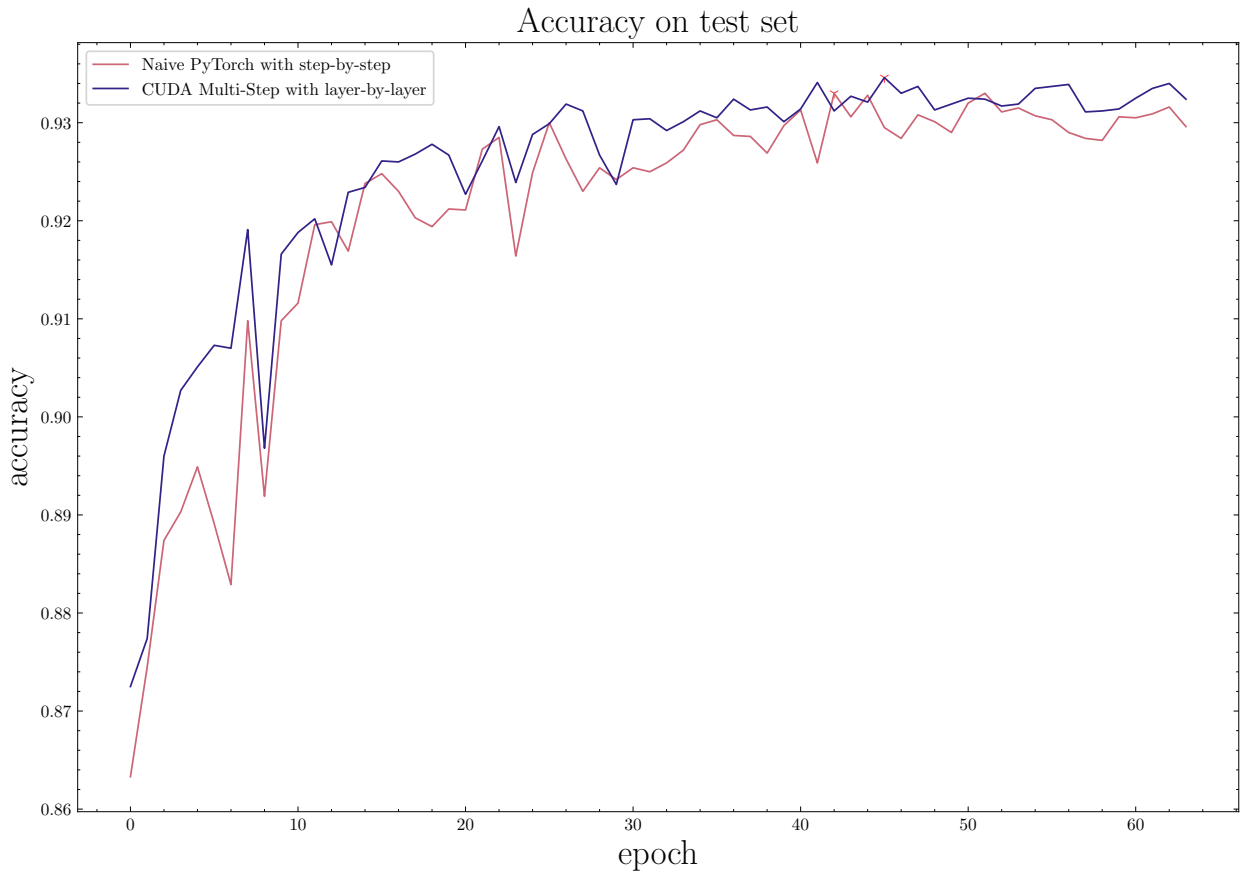
(续下页)

(接上页)

```
↪010720000202953816, test_acc=0.9324, max_test_acc=0.9346, total_time=11.
↪128222703933716
```

We get 93.46% accuracy, which is very close to 93.3% in 使用 *CUDA* 增强的神经元与逐层传播进行加速. Here are training logs:





In fact, we set an identical seed in both examples, but get a different results, which maybe caused by the numerical errors between cupy and pytorch functions. It can be found that the training execution time with cupy backend is 69% of the naive PyTorch SNN.

7.1.13 Neuromorphic Datasets Processing

Authors: fangwei123456

`spikingjelly.datasets` provides frequently-used neuromorphic datasets, including N-MNIST[1], CIFAR10-DVS[2], DVS128 Gesture[3], N-Caltech101[Page 251, 1], ASLDVS[4], etc. All datasets are processed by SpikingJelly in the same method, which is friendly for developers to write codes for new datasets. 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] 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 Automatically/Manually

SpikingJelly can download some datasets (e.g., CIFAR10-DVS) automatically. When we firstly use these datasets, SpikingJelly will download the dataset to `download` in the root directory. The `downloadable()` function of each dataset defines whether this dataset can be downloaded automatically, and the `resource_url_md5()` function defines the download url and MD5 of each file. Here is an example:

```
from spikingjelly.datasets.cifar10_dvs import CIFAR10DVS
from spikingjelly.datasets.dvs128_gesture import DVS128Gesture

print('CIFAR10-DVS downloadable', CIFAR10DVS.downloadable())
print('resource, url, md5/n', CIFAR10DVS.resource_url_md5())

print('DVS128Gesture downloadable', DVS128Gesture.downloadable())
print('resource, url, md5/n', DVS128Gesture.resource_url_md5())
```

The outputs are:

```
CIFAR10-DVS downloadable True
resource, url, md5
 [('airplane.zip', 'https://ndownloader.figshare.com/files/7712788',
↪'0afd5c4bf9ae06af762a77b180354fdd'), ('automobile.zip', 'https://ndownloader.
↪figshare.com/files/7712791', '8438dfeba3bc970c94962d995b1b9bdd'), ('bird.zip',
↪'https://ndownloader.figshare.com/files/7712794', 'a9c207c91c55b9dc2002dc21c684d785
↪'), ('cat.zip', 'https://ndownloader.figshare.com/files/7712812',
↪'52c63c677c2b15fa5146a8daf4d56687'), ('deer.zip', 'https://ndownloader.figshare.com/
↪files/7712815', 'b6bf21f6c04d21ba4e23fc3e36c8a4a3'), ('dog.zip', 'https://
↪ndownloader.figshare.com/files/7712818', 'f379ebdf6703d16e0a690782e62639c3'), (
↪'frog.zip', 'https://ndownloader.figshare.com/files/7712842',
↪'cad6ed91214b1c7388a5f6ee56d08803'), ('horse.zip', 'https://ndownloader.figshare.
↪com/files/7712851', 'e7cbbf77bec584ffbf913f00e682782a'), ('ship.zip', 'https://
↪ndownloader.figshare.com/files/7712836', '41c7bd7d6b251be82557c6cce9a7d5c9'), (
↪'truck.zip', 'https://ndownloader.figshare.com/files/7712839',
↪'89f3922fd147d9aeff89e76a2b0b70a7')]
DVS128Gesture downloadable False
resource, url, md5
 [('DvsGesture.tar.gz', 'https://ibm.ent.box.com/s/3hiq58ww1pbbjrinh367ykfdf60xsfm8/
↪folder/50167556794', '8a5c71fb11e24e5ca5b11866ca6c00a1'), ('gesture_mapping.csv',
↪'https://ibm.ent.box.com/s/3hiq58ww1pbbjrinh367ykfdf60xsfm8/folder/50167556794',
↪'109b2ae64a0e1f3ef535b18ad7367fd1'), ('LICENSE.txt', 'https://ibm.ent.box.com/s/
↪3hiq58ww1pbbjrinh367ykfdf60xsfm8/folder/50167556794',
↪'065e10099753156f18f51941e6e44b66'), ('README.txt', 'https://ibm.ent.box.com/s/
↪3hiq58ww1pbbjrinh367ykfdf60xsfm8/folder/50167556794',
↪'a0663d3b1d8307c329a43d949ee32d19')]
```

The DVS128 Gesture dataset can not be downloaded automatically. But its `resource_url_md5()` will tell user where to download. 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/download`, then the directory structure is

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

Get Events Data

Let us create train set. We set `data_type='event'` to use Event data rather than frame data.

```
from spikingjelly.datasets.dvs128_gesture import DVS128Gesture

root_dir = 'D:/datasets/DVS128Gesture'
train_set = DVS128Gesture(root_dir, train=True, data_type='event')
```

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:

```
The [D:/datasets/DVS128Gesture/download] directory for saving downloaed files already␣
↪exists, check files...
Mkdir [D:/datasets/DVS128Gesture/extract].
Extract [D:/datasets/DVS128Gesture/download/DvsGesture.tar.gz] to [D:/datasets/
↪DVS128Gesture/extract].
Mkdir [D:/datasets/DVS128Gesture/events_np].
Start to convert the origin data from [D:/datasets/DVS128Gesture/extract] to [D:/
↪datasets/DVS128Gesture/events_np] in np.ndarray format.
Mkdir [('D:/datasets/DVS128Gesture//events_np//train', 'D:/datasets/DVS128Gesture//
↪events_np//test').
Mkdir ['0', '1', '10', '2', '3', '4', '5', '6', '7', '8', '9'] in [D:/datasets/
↪DVS128Gesture/events_np/train] and ['0', '1', '10', '2', '3', '4', '5', '6', '7', '8
```

(续下页)

(接上页)

```
↪', '9'] in [D:/datasets/DVS128Gesture/events_np/test].
Start the ThreadPoolExecutor with max workers = [8].
Start to split [D:/datasets/DVS128Gesture/extract/DvsGesture/user02_fluorescent.
↪aedat] to samples.
[D:/datasets/DVS128Gesture/events_np/train/0/user02_fluorescent_0.npz] saved.
[D:/datasets/DVS128Gesture/events_np/train/1/user02_fluorescent_0.npz] saved.

......

[D:/datasets/DVS128Gesture/events_np/test/8/user29_lab_0.npz] saved.
[D:/datasets/DVS128Gesture/events_np/test/9/user29_lab_0.npz] saved.
[D:/datasets/DVS128Gesture/events_np/test/10/user29_lab_0.npz] saved.
Used time = [1017.27s].
All aedat files have been split to samples and saved into [('D:/datasets/
↪DVS128Gesture//events_np//train', 'D:/datasets/DVS128Gesture//events_np//test')].
```

We have to wait for a moment because the cutting and extracting is very slow. A `events_np` folder will be created and contain the train/test set:

```
|-- events_np
|   |-- test
|   `-- train
```

Print a sample:

```
event, label = train_set[0]
for k in event.keys():
    print(k, event[k])
print('label', label)
```

The output is:

```
t [80048267 80048277 80048278 ... 85092406 85092538 85092700]
x [49 55 55 ... 60 85 45]
y [82 92 92 ... 96 86 90]
p [1 0 0 ... 1 0 0]
label 0
```

where `event` is a dictionary with keys `['t', 'x', 'y', 'p']`;"label" 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[5] 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 $F(j)$ 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, data_type='frame', frames_number=20,␣
↪split_by='number')
```

The outputs from the terminal are:

```
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/0].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/1].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/10].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/2].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/3].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/4].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/5].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/6].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/7].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/8].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/test/9].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/0].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/1].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/10].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/2].
```

(续下页)

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

(接上页)

```
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/3].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/4].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/5].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/6].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/7].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/8].
Mkdir [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/9].
Start ThreadPoolExecutor with max workers = [8].
Start to integrate [D:/datasets/DVS128Gesture/events_np/test/0/user24_fluorescent_0.
↪npz] to frames and save to [D:/datasets/DVS128Gesture/frames_number_20_split_by_
↪number/test/0].
Start to integrate [D:/datasets/DVS128Gesture/events_np/test/0/user24_fluorescent_led_
↪0.npz] to frames and save to [D:/datasets/DVS128Gesture/frames_number_20_split_by_
↪number/test/0].

......

Frames [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/train/9/user23_lab_
↪0.npz] saved.Frames [D:/datasets/DVS128Gesture/frames_number_20_split_by_number/
↪train/9/user23_led_0.npz] saved.

Used time = [102.11s].
```

A `frames_number_20_split_by_number` folder will be created and contain the Frame data.

Print a sample:

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

The outputs are:

```
(20, 2, 128, 128)
```

Let us visualize a sample:

```
from spikingjelly.datasets import play_frame
frame, label = train_set[500]
play_frame(frame)
```

We will get the images like:

Fixed Duration Integrating

Integrating by fixed duration is more compatible with the practical application. For example, if we set duration as `10 ms`, then a sample with length `L ms` can be integrated to frames with frame number `math.floor(L / 10)`. However, the lengthes of samples in neuromorphic datasets are not identical, and we will get frames with different frame numbers when integrating with fixed duration. Fortunately, we can use *spikingjelly.datasets.pad_sequence_collate* and *spikingjelly.datasets.padded_sequence_mask* to pad/unpad frames.

Example codes:

```
import torch
from torch.utils.data import DataLoader
from spikingjelly.datasets import pad_sequence_collate, padded_sequence_mask, dvs128_
↪gesture
root='D:/datasets/DVS128Gesture'
train_set = dvs128_gesture.DVS128Gesture(root, data_type='frame', duration=1000000,␣
↪train=True)
for i in range(5):
    x, y = train_set[i]
    print(f'x[{i}].shape=[T, C, H, W]={x.shape}')
train_data_loader = DataLoader(train_set, collate_fn=pad_sequence_collate, batch_
↪size=5)
for x, y, x_len in train_data_loader:
    print(f'x.shape=[N, T, C, H, W]={tuple(x.shape)}')
    print(f'x_len={x_len}')
    mask = padded_sequence_mask(x_len)  # mask.shape = [T, N]
    print(f'mask=\n{mask.t().int()}')
    break
```

The outputs are:

```
The directory [D:/datasets/DVS128Gesture\duration_1000000] already exists.
x[0].shape=[T, C, H, W]=(6, 2, 128, 128)
x[1].shape=[T, C, H, W]=(6, 2, 128, 128)
x[2].shape=[T, C, H, W]=(5, 2, 128, 128)
x[3].shape=[T, C, H, W]=(5, 2, 128, 128)
x[4].shape=[T, C, H, W]=(7, 2, 128, 128)
x.shape=[N, T, C, H, W]=(5, 7, 2, 128, 128)
x_len=tensor([6, 6, 5, 5, 7])
mask=
tensor([[1, 1, 1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1, 1, 1]], dtype=torch.int32)
```

Custom Integrating Method

SpikingJelly provides user-defined integrating method. The user should provide a function `custom_integrate_function` and the name of directory `custom_integrated_frames_dir_name` for saving frames.

`custom_integrate_function` is a user-defined function that inputs are `events, H, W`. `events` is a dict whose keys are `['t', 'x', 'y', 'p']` and values are `numpy.ndarray`. `H` is the height of the data and `W` is the weight of the data. For example, H=128 and W=128 for the DVS128 Gesture dataset. The function should return frames.

`custom_integrated_frames_dir_name` can be `None`, and then the the name of directory for saving frames will be set to `custom_integrate_function.__name__`.

For example, if we want to split events to two parts randomly, and integrate two parts to two frames, we can define such a function:

```
import spikingjelly.datasets as sjds
def integrate_events_to_2_frames_randomly(events: Dict, H: int, W: int):
    index_split = np.random.randint(low=0, high=events['t'].__len__())
    frames = np.zeros([2, 2, H, W])
    t, x, y, p = (events[key] for key in ('t', 'x', 'y', 'p'))
    frames[0] = sjds.integrate_events_segment_to_frame(x, y, p, H, W, 0, index_split)
    frames[1] = sjds.integrate_events_segment_to_frame(x, y, p, H, W, index_split,␣
↪events['t'].__len__())
    return frames
```

Now let us use this function to create frames dataset:

```
train_set = DVS128Gesture(root_dir, train=True, data_type='frame', custom_integrate_
↪function=integrate_events_to_2_frames_randomly)
```

After the process finished, there will be a `integrate_events_to_2_frames_randomly` directory in `root_dir`. And the `integrate_events_to_2_frames_randomly` directory will save our frames integrated by the custom integrating function.

Now let us visualize the frames:

```
from spikingjelly.datasets import play_frame
frame, label = train_set[500]
play_frame(frame)
```

SpikingJelly provides more methods to integrate events to frames. Read the API doc for more details.

7.1.14 Classify DVS128 Gesture

Author: fangwei123456

We have learned how to use neuromorphic datasets in last tutorial *Neuromorphic Datasets Processing*. Now, let us start to build a SNN to classify the DVS128 Gesture dataset. We will use the SNN from Incorporating Learnable Membrane Time Constant to Enhance Learning of Spiking Neural Networks[1]. We will use LIF neurons and max pooling in this

[1] Fang, Wei, et al. "Incorporating learnable membrane time constant to enhance learning of spiking neural networks." Proceedings of the IEEE/CVF International Conference on Computer Vision. 2021.

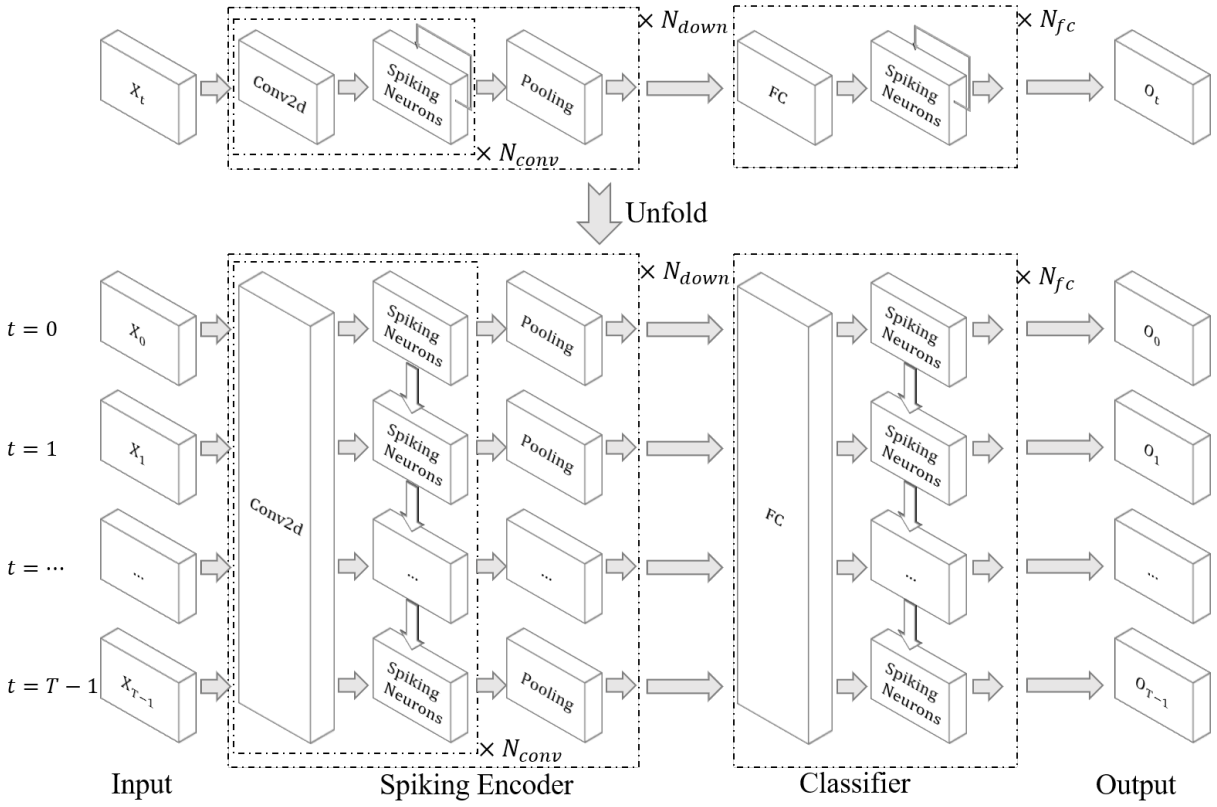
SNN.

The paper[Page 260, 1] uses an old version of SpikingJelly. The origin codes and logs are available at: Parametric-Leaky-Integrate-and-Fire-Spiking-Neuron

In this tutorial, we will write codes by the new version of SpikingJelly, and our codes run faster than the origin codes.

Define The Network

The paper[Page 260, 1] use a general structure to build SNNs for different datasets, which is shown in the following figure:



$N_{conv} = 1, N_{down} = 5, N_{fc} = 2$ for the DVS128 Gesture dataset.

The detailed network structure is *{c128k3s1-BN-LIF-MPk2s2}*5-DP-FC512-LIF-DP-FC110-LIF-APk10s10}*, where *APk10s10* is an additional voting layer.

The meanings of symbol are:

c128k3s1: `torch.nn.Conv2d(in_channels, out_channels=128, kernel_size=3, padding=1)`

BN: `torch.nn.BatchNorm2d(128)`

MPk2s2: `torch.nn.MaxPool2d(2, 2)`

DP: `spikingjelly.clock_driven.layer.Dropout(0.5)`

FC512: `torch.nn.Linear(in_features, out_features=512`

APk10s10: `torch.nn.AvgPool1d(2, 2)`

For simplicity, we firstly implement the network by the step-by-step mode:

```
class VotingLayer(nn.Module):
    def __init__(self, voter_num: int):
        super().__init__()
        self.voting = nn.AvgPool1d(voter_num, voter_num)
    def forward(self, x: torch.Tensor):
        # x.shape = [N, voter_num * C]
        # ret.shape = [N, C]
        return self.voting(x.unsqueeze(1)).squeeze(1)

class PythonNet(nn.Module):
    def __init__(self, channels: int):
        super().__init__()
        conv = []
        conv.extend(PythonNet.conv3x3(2, channels))
        conv.append(nn.MaxPool2d(2, 2))
        for i in range(4):
            conv.extend(PythonNet.conv3x3(channels, channels))
            conv.append(nn.MaxPool2d(2, 2))
        self.conv = nn.Sequential(*conv)
        self.fc = nn.Sequential(
            nn.Flatten(),
            layer.Dropout(0.5),
            nn.Linear(channels * 4 * 4, channels * 2 * 2, bias=False),
            neuron.LIFNode(tau=2.0, surrogate_function=surrogate.ATan(), detach_
↪reset=True),
            layer.Dropout(0.5),
            nn.Linear(channels * 2 * 2, 110, bias=False),
            neuron.LIFNode(tau=2.0, surrogate_function=surrogate.ATan(), detach_
↪reset=True)
        )
        self.vote = VotingLayer(10)

    @staticmethod
    def conv3x3(in_channels: int, out_channels):
        return [
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1,␣
↪bias=False),
            nn.BatchNorm2d(out_channels),
            neuron.LIFNode(tau=2.0, surrogate_function=surrogate.ATan(), detach_
↪reset=True)
```

(续下页)

(接上页)

```
        ]
```

Forward and Loss

We set simulating time-steps as `T`, batch size as `N`, then the frames `x` from `DataLoader` will have `x.shape=[N, T, 2, 128, 128]`. We firstly convert `x` to `shape=[T, N, 2, 128, 128]`.

Then, we send `x[t]` to the network, accumulate the output spikes and get the firing rate `out_spikes / x.shape[0]`, which is a tensor with `shape=[N, 11]`.

```
def forward(self, x: torch.Tensor):
    x = x.permute(1, 0, 2, 3, 4)  # [N, T, 2, H, W] -> [T, N, 2, H, W]
    out_spikes = self.vote(self.fc(self.conv(x[0])))
    for t in range(1, x.shape[0]):
        out_spikes += self.vote(self.fc(self.conv(x[t])))
    return out_spikes / x.shape[0]
```

The loss is defined by the MSE between firing rate and the label in one hot format:

```
for frame, label in train_data_loader:
    optimizer.zero_grad()
    frame = frame.float().to(args.device)
    label = label.to(args.device)
    label_onehot = F.one_hot(label, 11).float()

    out_fr = net(frame)
    loss = F.mse_loss(out_fr, label_onehot)
    loss.backward()
    optimizer.step()

    functional.reset_net(net)
```

Accelerate by CUDA Neurons and Layer-by-layer

If the reader is not familiar with propagation pattern in SpikingJelly, please read the previous tutorials: *Propagation Pattern* and *Accelerate with CUDA-Enhanced Neuron and Layer-by-Layer Propagation*.

We have built the net in the step-by-step model, whose codes are user-friendly but run slower. Now let us re-write the net in the layer-by-layer mode with CUDA neurons:

```
import cupy

class CextNet(nn.Module):
```

(续下页)

(接上页)

```
    def __init__(self, channels: int):
        super().__init__()
        conv = []
        conv.extend(CextNet.conv3x3(2, channels))
        conv.append(layer.SeqToANNContainer(nn.MaxPool2d(2, 2)))
        for i in range(4):
            conv.extend(CextNet.conv3x3(channels, channels))
            conv.append(layer.SeqToANNContainer(nn.MaxPool2d(2, 2)))
        self.conv = nn.Sequential(*conv)
        self.fc = nn.Sequential(
            nn.Flatten(2),
            layer.MultiStepDropout(0.5),
            layer.SeqToANNContainer(nn.Linear(channels * 4 * 4, channels * 2 * 2,␣
↪bias=False)),
            neuron.MultiStepLIFNode(tau=2.0, surrogate_function=surrogate.ATan(),␣
↪detach_reset=True, backend='cupy'),
            layer.MultiStepDropout(0.5),
            layer.SeqToANNContainer(nn.Linear(channels * 2 * 2, 110, bias=False)),
            neuron.MultiStepLIFNode(tau=2.0, surrogate_function=surrogate.ATan(),␣
↪detach_reset=True, backend='cupy')
        )
        self.vote = VotingLayer(10)

    def forward(self, x: torch.Tensor):
        x = x.permute(1, 0, 2, 3, 4)  # [N, T, 2, H, W] -> [T, N, 2, H, W]
        out_spikes = self.fc(self.conv(x))  # shape = [T, N, 110]
        return self.vote(out_spikes.mean(0))

    @staticmethod
    def conv3x3(in_channels: int, out_channels):
        return [
            layer.SeqToANNContainer(
                nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1,␣
↪bias=False),
                nn.BatchNorm2d(out_channels),
            ),
            neuron.MultiStepLIFNode(tau=2.0, surrogate_function=surrogate.ATan(),␣
↪detach_reset=True, backend='cupy')
        ]
```

We can find that the two kind of models are similar. All stateless layers, e,g, `Conv2d`, will be contained in `layer.SeqToANNContainer`. The forward function is defined easily:

```
def forward(self, x: torch.Tensor):
    x = x.permute(1, 0, 2, 3, 4)  # [N, T, 2, H, W] -> [T, N, 2, H, W]
    out_spikes = self.fc(self.conv(x))  # shape = [T, N, 110]
    return self.vote(out_spikes.mean(0))
```

Code Details

We add more arguments:

```
parser = argparse.ArgumentParser(description='Classify DVS128 Gesture')
parser.add_argument('-T', default=16, type=int, help='simulating time-steps')
parser.add_argument('-device', default='cuda:0', help='device')
parser.add_argument('-b', default=16, type=int, help='batch size')
parser.add_argument('-epochs', default=64, type=int, metavar='N',
                    help='number of total epochs to run')
parser.add_argument('-j', default=4, type=int, metavar='N',
                    help='number of data loading workers (default: 4)')
parser.add_argument('-channels', default=128, type=int, help='channels of Conv2d in␣
↪SNN')
parser.add_argument('-data_dir', type=str, help='root dir of DVS128 Gesture dataset')
parser.add_argument('-out_dir', type=str, help='root dir for saving logs and␣
↪checkpoint')

parser.add_argument('-resume', type=str, help='resume from the checkpoint path')
parser.add_argument('-amp', action='store_true', help='automatic mixed precision␣
↪training')
parser.add_argument('-cupy', action='store_true', help='use CUDA neuron and multi-
↪step forward mode')


parser.add_argument('-opt', type=str, help='use which optimizer. SDG or Adam')
parser.add_argument('-lr', default=0.001, type=float, help='learning rate')
parser.add_argument('-momentum', default=0.9, type=float, help='momentum for SGD')
parser.add_argument('-lr_scheduler', default='CosALR', type=str, help='use which␣
↪schedule. StepLR or CosALR')
parser.add_argument('-step_size', default=32, type=float, help='step_size for StepLR')
parser.add_argument('-gamma', default=0.1, type=float, help='gamma for StepLR')
parser.add_argument('-T_max', default=32, type=int, help='T_max for CosineAnnealingLR
↪')
```

Using automatic mixed precision (AMP) can accelerate training and reduce memory consumption:

```
if args.amp:
```

(续下页)

(接上页)

```
    with amp.autocast():
        out_fr = net(frame)
        loss = F.mse_loss(out_fr, label_onehot)
    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()
else:
    out_fr = net(frame)
    loss = F.mse_loss(out_fr, label_onehot)
    loss.backward()
    optimizer.step()
```

We can also resume from a check point:

```
#...........
if args.resume:
    checkpoint = torch.load(args.resume, map_location='cpu')
    net.load_state_dict(checkpoint['net'])
    optimizer.load_state_dict(checkpoint['optimizer'])
    lr_scheduler.load_state_dict(checkpoint['lr_scheduler'])
    start_epoch = checkpoint['epoch'] + 1
    max_test_acc = checkpoint['max_test_acc']
# ...

for epoch in range(start_epoch, args.epochs):
# train...

# test...

    checkpoint = {
        'net': net.state_dict(),
        'optimizer': optimizer.state_dict(),
        'lr_scheduler': lr_scheduler.state_dict(),
        'epoch': epoch,
        'max_test_acc': max_test_acc
    }

    # ...

    torch.save(checkpoint, os.path.join(out_dir, 'checkpoint_latest.pth'))
```

Star Training

The complete codes are available at spikingjelly.clock_driven.examples.classify_dvsg.

We train the net in a linux server with *Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz* CPU and *GeForce RTX 2080 Ti* GPU. We use almost the same hyper-parameters with those in the paper[Page 260, 1] with little difference, which is we use `T=16` because our *GeForce RTX 2080 Ti* only has 12GB memory, while the paper uses `T=20`. Besides, we use AMP to accelerate, which may cause slightly worse accuracy than the full precision training.

Let us try to train the step-by-step network:

```
(test-env) root@de41f92009cf3011eb0ac59057a81652d2d0-fangw1714-0:/userhome/test#␣
↪python -m spikingjelly.clock_driven.examples.classify_dvsg -data_dir /userhome/
↪datasets/DVS128Gesture -out_dir ./logs -amp -opt Adam -device cuda:0 -lr_scheduler␣
↪CosALR -T_max 64 -epochs 256
Namespace(T=16, T_max=64, amp=True, b=16, cupy=False, channels=128, data_dir='/
↪userhome/datasets/DVS128Gesture', device='cuda:0', epochs=256, gamma=0.1, j=4, lr=0.
↪001, lr_scheduler='CosALR', momentum=0.9, opt='Adam', out_dir='./logs', resume=None,
↪ step_size=32)
PythonNet(
  (conv): Sequential(
    (0): Conv2d(2, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
    (2): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
    (6): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
    (9): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
    (10): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
```

(续下页)

(接上页)

```
    )
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (12): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
    (13): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
    (14): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (16): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
    (17): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
    (18): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (19): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Dropout(p=0.5)
    (2): Linear(in_features=2048, out_features=512, bias=False)
    (3): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
    (4): Dropout(p=0.5)
    (5): Linear(in_features=512, out_features=110, bias=False)
    (6): LIFNode(
      v_threshold=1.0, v_reset=0.0, tau=2.0
      (surrogate_function): ATan(alpha=2.0, spiking=True)
    )
  )
  (vote): VotingLayer(
    (voting): AvgPool1d(kernel_size=(10,), stride=(10,), padding=(0,))
  )
)
The directory [/userhome/datasets/DVS128Gesture/frames_number_16_split_by_number]␣
↪already exists.
The directory [/userhome/datasets/DVS128Gesture/frames_number_16_split_by_number]␣
```

(续下页)

(接上页)

```
↪already exists.
Mkdir ./logs/T_16_b_16_c_128_Adam_lr_0.001_CosALR_64_amp.
Namespace(T=16, T_max=64, amp=True, b=16, cupy=False, channels=128, data_dir='/
↪userhome/datasets/DVS128Gesture', device='cuda:0', epochs=256, gamma=0.1, j=4, lr=0.
↪001, lr_scheduler='CosALR', momentum=0.9, opt='Adam', out_dir='./logs', resume=None,
↪ step_size=32)
epoch=0, train_loss=0.06680945929599134, train_acc=0.4032534246575342, test_loss=0.
↪04891310722774102, test_acc=0.6180555555555556, max_test_acc=0.6180555555555556,␣
↪total_time=27.759592294692993
```

It takes 27.76s to finish an epoch. We stop it and train the faster network:

```
(test-env) root@de41f92009cf3011eb0ac59057a81652d2d0-fangw1714-0:/userhome/test#␣
↪python -m spikingjelly.clock_driven.examples.classify_dvsg -data_dir /userhome/
↪datasets/DVS128Gesture -out_dir ./logs -amp -opt Adam -device cuda:0 -lr_scheduler␣
↪CosALR -T_max 64 -cupy -epochs 256
Namespace(T=16, T_max=64, amp=True, b=16, cupy=True, channels=128, data_dir='/
↪userhome/datasets/DVS128Gesture', device='cuda:0', epochs=256, gamma=0.1, j=4, lr=0.
↪001, lr_scheduler='CosALR', momentum=0.9, opt='Adam', out_dir='./logs', resume=None,
↪ step_size=32)
CextNet(
  (conv): Sequential(
    (0): SeqToANNContainer(
      (module): Sequential(
        (0): Conv2d(2, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
↪function=ATan, alpha=2.0 tau=2.0)
    (2): SeqToANNContainer(
      (module): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
    )
    (3): SeqToANNContainer(
      (module): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
```

(续下页)

(接上页)

```
    (4): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
↪function=ATan, alpha=2.0 tau=2.0)
    (5): SeqToANNContainer(
      (module): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
    )
    (6): SeqToANNContainer(
      (module): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (7): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
↪function=ATan, alpha=2.0 tau=2.0)
    (8): SeqToANNContainer(
      (module): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
    )
    (9): SeqToANNContainer(
      (module): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (10): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
↪function=ATan, alpha=2.0 tau=2.0)
    (11): SeqToANNContainer(
      (module): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
    )
    (12): SeqToANNContainer(
      (module): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (13): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
```

(续下页)

(接上页)

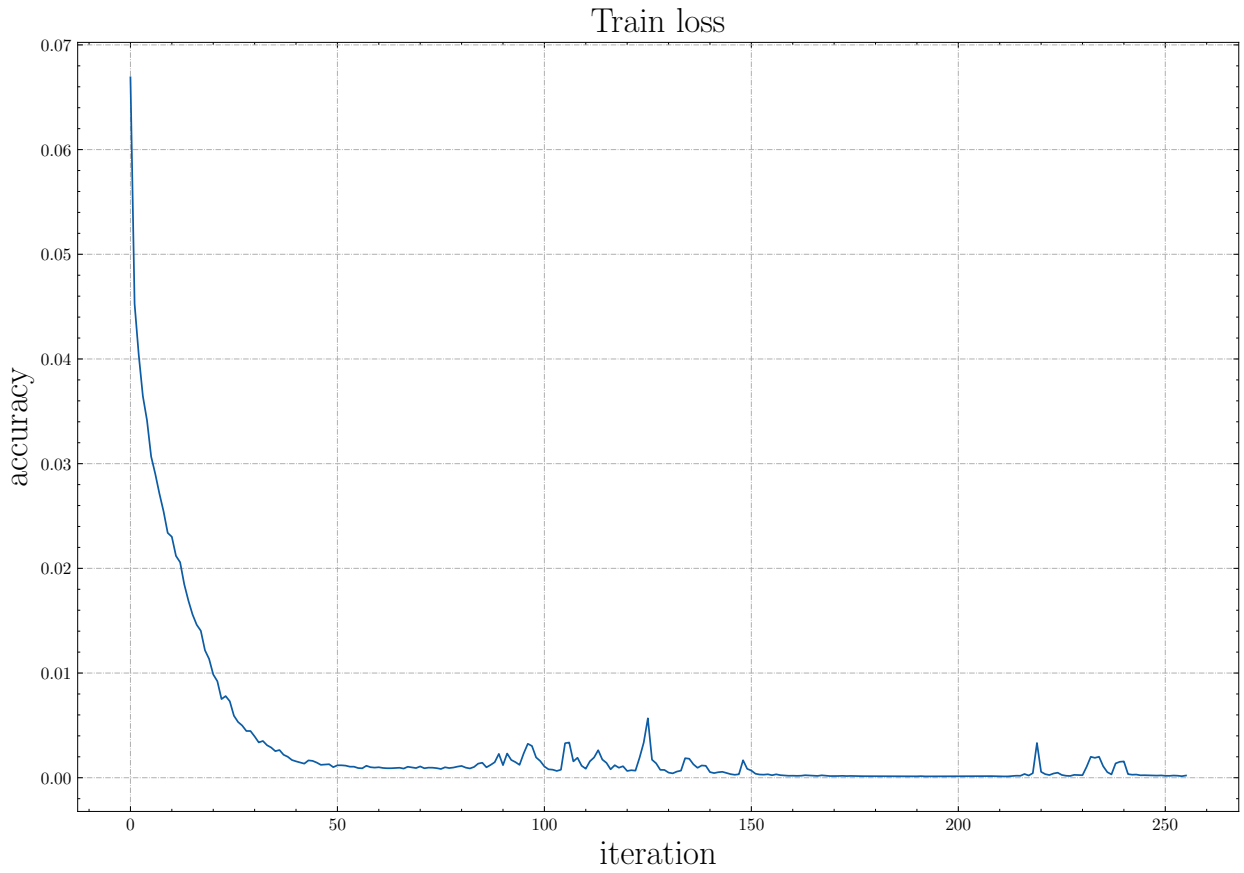
```
↪function=ATan, alpha=2.0 tau=2.0)
    (14): SeqToANNContainer(
      (module): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
↪mode=False)
    )
  )
  (fc): Sequential(
    (0): Flatten(start_dim=2, end_dim=-1)
    (1): MultiStepDropout(p=0.5)
    (2): SeqToANNContainer(
      (module): Linear(in_features=2048, out_features=512, bias=False)
    )
    (3): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
↪function=ATan, alpha=2.0 tau=2.0)
    (4): MultiStepDropout(p=0.5)
    (5): SeqToANNContainer(
      (module): Linear(in_features=512, out_features=110, bias=False)
    )
    (6): MultiStepLIFNode(v_threshold=1.0, v_reset=0.0, detach_reset=True, surrogate_
↪function=ATan, alpha=2.0 tau=2.0)
  )
  (vote): VotingLayer(
    (voting): AvgPool1d(kernel_size=(10,), stride=(10,), padding=(0,))
  )
)
The directory [/userhome/datasets/DVS128Gesture/frames_number_16_split_by_number]␣
↪already exists.
The directory [/userhome/datasets/DVS128Gesture/frames_number_16_split_by_number]␣
↪already exists.
Mkdir ./logs/T_16_b_16_c_128_Adam_lr_0.001_CosALR_64_amp_cupy.
Namespace(T=16, T_max=64, amp=True, b=16, cupy=True, channels=128, data_dir='/
↪userhome/datasets/DVS128Gesture', device='cuda:0', epochs=256, gamma=0.1, j=4, lr=0.
↪001, lr_scheduler='CosALR', momentum=0.9, opt='Adam', out_dir='./logs', resume=None,
↪ step_size=32)
epoch=0, train_loss=0.06690179117738385, train_acc=0.4092465753424658, test_loss=0.
↪049108295158172645, test_acc=0.6145833333333334, max_test_acc=0.6145833333333334,␣
↪total_time=18.169376373291016

...

Namespace(T=16, T_max=64, amp=True, b=16, cupy=True, channels=128, data_dir='/
↪userhome/datasets/DVS128Gesture', device='cuda:0', epochs=256, gamma=0.1, j=4, lr=0.
↪001, lr_scheduler='CosALR', momentum=0.9, opt='Adam', out_dir='./logs', resume=None,
```

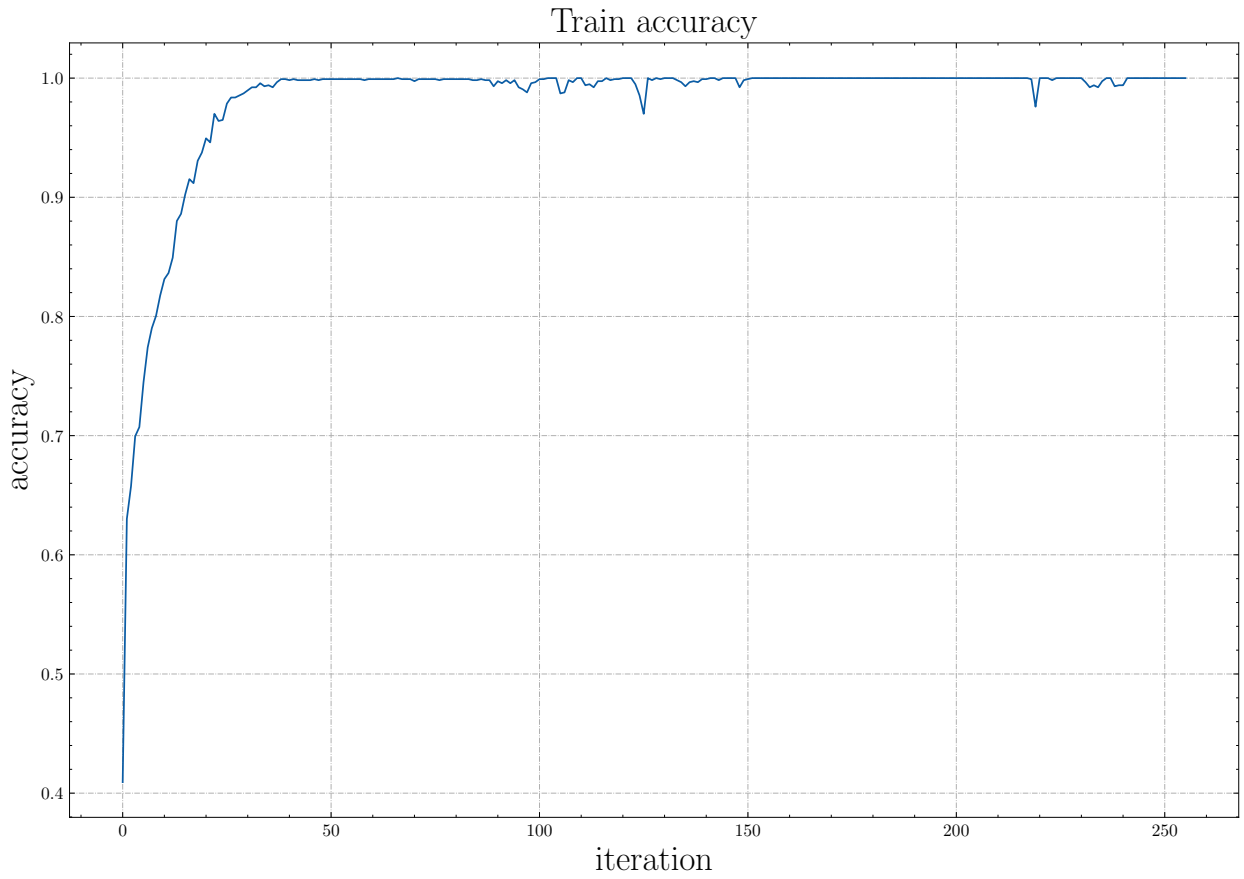
(续下页)

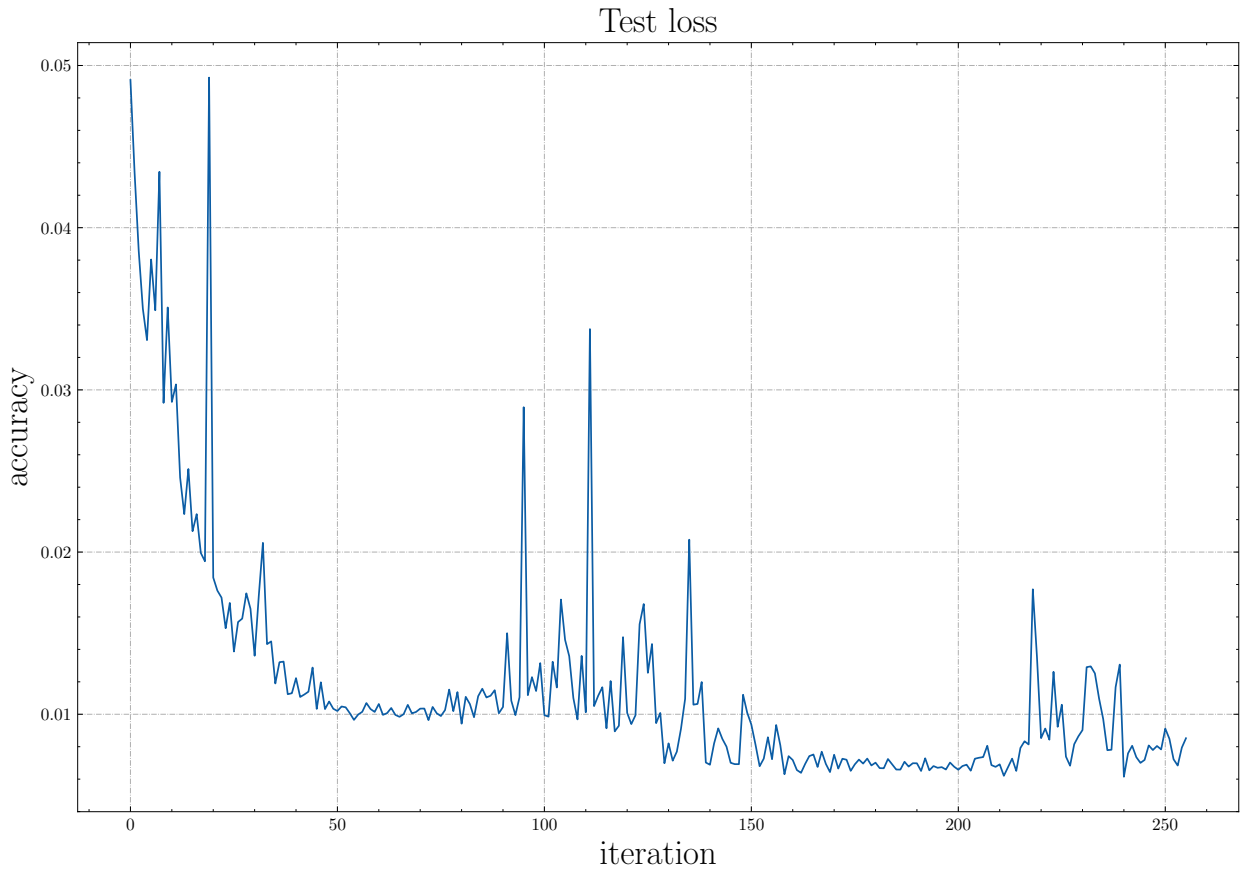
(接上页)

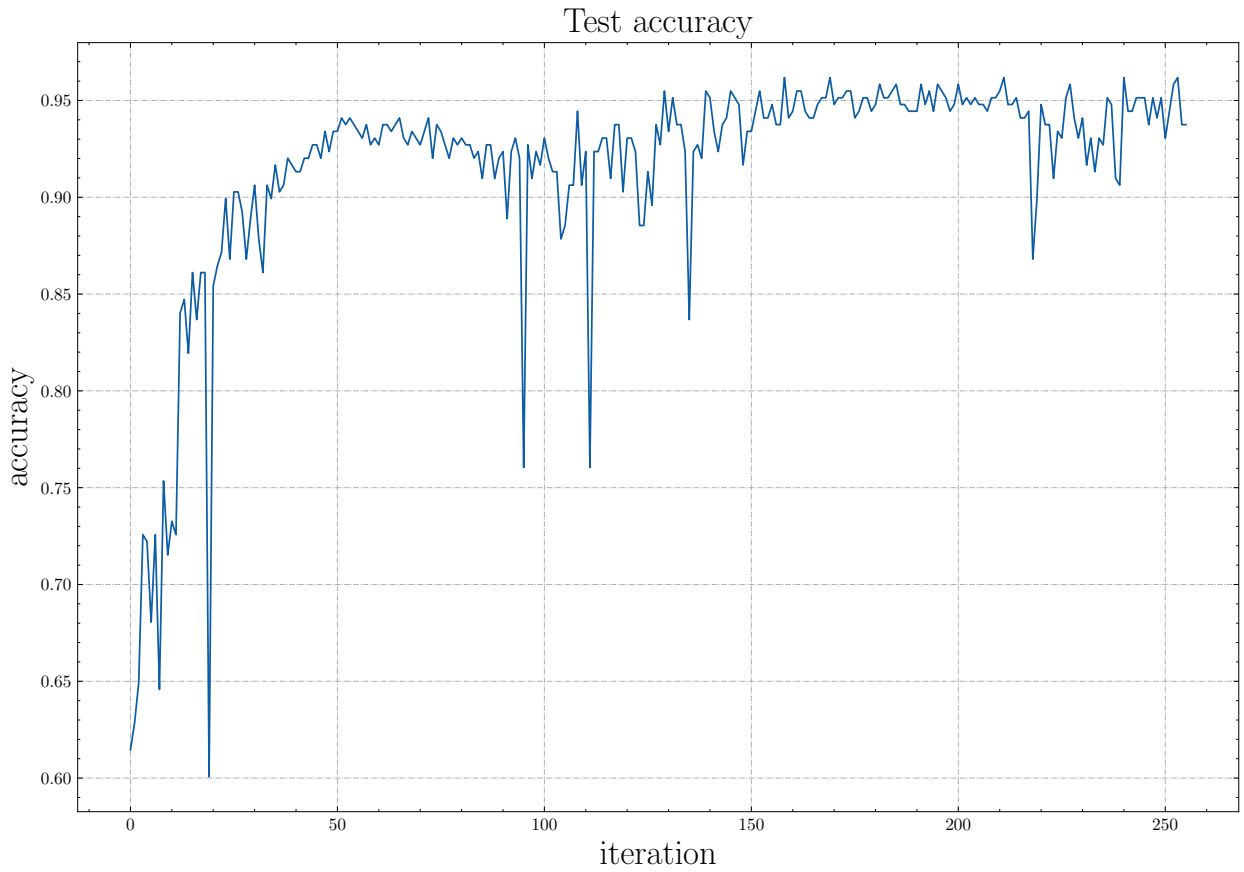
```
↪ step_size=32)
epoch=255, train_loss=0.00021228195577325645, train_acc=1.0, test_loss=0.
↪00852209396485576, test_acc=0.9375, max_test_acc=0.9618055555555556, total_time=17.
↪49005389213562
```

It takes 18.17s to finish an epoch, which is much faster. After 256 epochs, we will get the maximum accuracy 96.18%. The logs curves during training are:




Train accuracy
accuracy
iteration
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
50
100
150
200
250


Test loss
accuracy
iteration
0.05
0.04
0.03
0.02
0.01
0
50
100
150
200
250



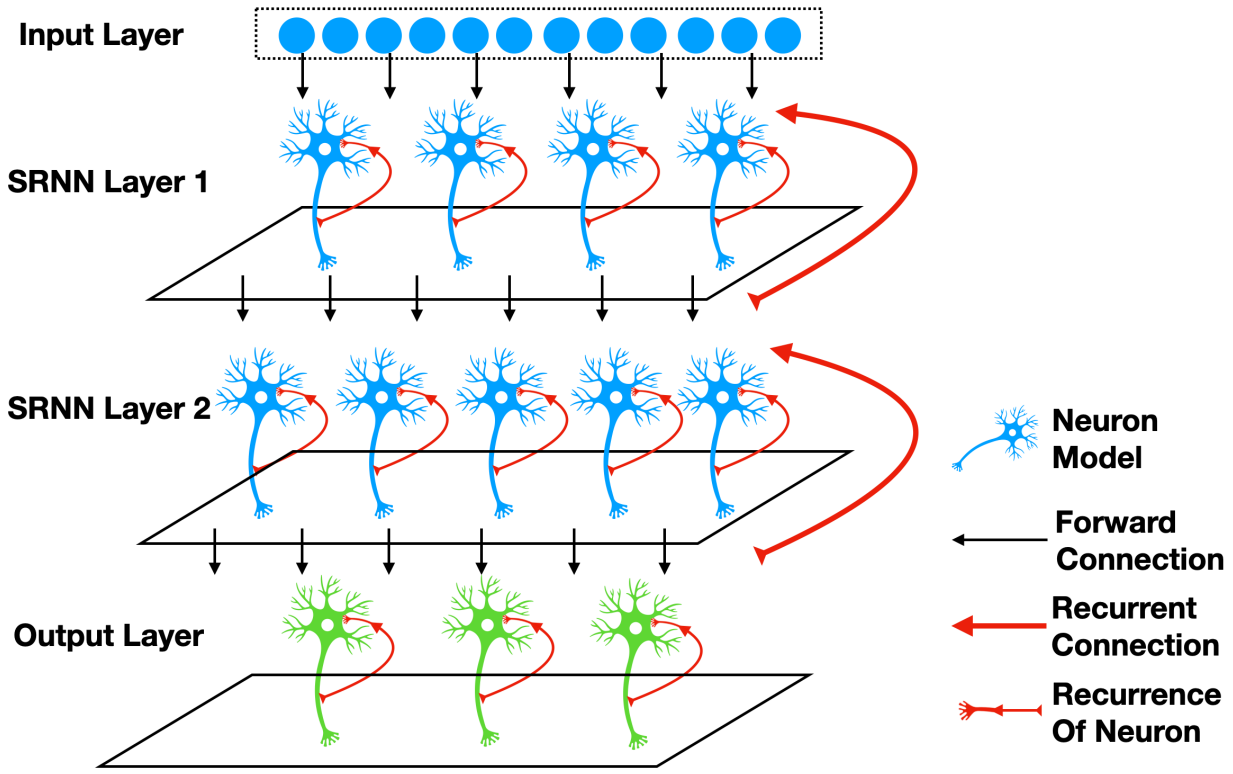
7.1.15 Recurrent Connections and Stateful Synapses

Author: fangwei123456

Recurrent Connections

The recurrent connections connect a module’ s outputs to its inputs. For example,[1] uses a SRNN(recurrent networks of spiking neurons), which is shown in the following figure:

[1] 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.



It is easy to use SpikingJelly to implement the recurrent module. Considering a simple case that we add a connection to make the neuron's outputs $s[t]$ at time-step t can add with external inputs $x[t+1]$ at time-step $t+1$. It can be implemented by *spikingjelly.clock_driven.layer.ElementWiseRecurrentContainer*. `ElementWiseRecurrentContainer` is a container that add a recurrent connection to the contained `sub_module`. The connection is a user-defined element-wise function $z = f(x, y)$. Denote the inputs and outputs of `sub_module` as $i[t]$ and $y[t]$ (Note that $y[t]$ is also the outputs of this module), and the inputs of this module as $x[t]$, then

$$i[t] = f(x[t], y[t-1])$$

where f is the user-defined element-wise function. We set $y[-1] = 0$.

Let us use `ElementWiseRecurrentContainer` to contain a IF neuron, and set the element-wise function as *add*:

$$i[t] = x[t] + y[t-1].$$

We use soft reset, and give the inputs as $x[t] = [1.5, 0, ..., 0]$:

```
T = 8
def element_wise_add(x, y):
    return x + y
net = ElementWiseRecurrentContainer(neuron.IFNode(v_reset=None), element_wise_add)
print(net)
x = torch.zeros([T])
```

(续下页)

(接上页)

```
x[0] = 1.5
for t in range(T):
    print(t, f'x[t]={x[t]}, s[t]={net(x[t])}')

functional.reset_net(net)
```

The outputs are:

```
ElementWiseRecurrentContainer(
  element-wise function=<function element_wise_add at 0x000001FE0F7968B0>
  (sub_module): IFNode(
    v_threshold=1.0, v_reset=None, detach_reset=False
    (surrogate_function): Sigmoid(alpha=1.0, spiking=True)
  )
)
0 x[t]=1.5, s[t]=1.0
1 x[t]=0.0, s[t]=1.0
2 x[t]=0.0, s[t]=1.0
3 x[t]=0.0, s[t]=1.0
4 x[t]=0.0, s[t]=1.0
5 x[t]=0.0, s[t]=1.0
6 x[t]=0.0, s[t]=1.0
7 x[t]=0.0, s[t]=1.0
```

We can find that due to the recurrent connection, even if $x[t] = 0$ when $t \geq 1$, the neuron can still fire because its output spike is fed back as input.

We can use *spikingjelly.clock_driven.layer.LinearRecurrentContainer* to implement a more complex recurrent connections.

Stateful Synapses

There are many papers using stateful synapses, e.g.,[2][3]. We can put *spikingjelly.clock_driven.layer.SynapseFilter* after a stateless synapse to get the stateful synapse:

```
stateful_conv = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=3, padding=1, stride=1),
    SynapseFilter(tau=100, learnable=True)
)
```

[2] Diehl P U, Cook M. Unsupervised learning of digit recognition using spike-timing-dependent plasticity[J]. Frontiers in computational neuroscience, 2015, 9: 99.

[3] Fang H, Shrestha A, Zhao Z, et al. Exploiting Neuron and Synapse Filter Dynamics in Spatial Temporal Learning of Deep Spiking Neural Network[J].

Ablation Study On Sequential FashionMNIST

Now we do a smple exmperiment on Sequential FashionMNIST to check whether recurrent connections and stateful synapses can promote the network’s temporal information fitting ability. Sequential FashionMNIST is using FashionMNIST as input row-by-row

> or column-by-column, rather than the whole image. Consequentially, the network classify Sequential FashionMNIST correctly

only when it can learn long-term dependencies. We will feed the image column-by-column, which is same with reading texts from left to right. Here is the example:

The following gif shows the column being read:

First, let us import packages:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.datasets
from spikingjelly.clock_driven.model import train_classify
from spikingjelly.clock_driven import neuron, surrogate, layer
from spikingjelly.clock_driven.functional import seq_to_ann_forward
from torchvision import transforms
import os, argparse

try:
    import cupy
    backend = 'cupy'
except ImportError:
    backend = 'torch'
```

Now let us define a plain feedforward network `Net`:

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28, 32)
        self.sn1 = neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), detach_
↪reset=True, backend=backend)
        self.fc2 = nn.Linear(32, 10)
        self.sn2 = neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), detach_
```

(续下页)

(接上页)

```
↪reset=True, backend=backend)

    def forward(self, x: torch.Tensor):
        # x.shape = [N, C, H, W]
        x.squeeze_(1)  # [N, H, W]
        x = x.permute(2, 0, 1)  # [W, N, H]
        x = seq_to_ann_forward(x, self.fc1)
        x = self.sn1(x)
        x = seq_to_ann_forward(x, self.fc2)
        x = self.sn2(x)
        return x.mean(0)
```

We add *spikingjelly.clock_driven.layer.SynapseFilter* after the first spiking neurons layer and get `StatefulSynapseNet`:

```
class StatefulSynapseNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28, 32)
        self.sn1 = neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), detach_
↪reset=True, backend=backend)
        self.sy1 = layer.MultiStepContainer(layer.SynapseFilter(tau=2.,␣
↪learnable=True))
        self.fc2 = nn.Linear(32, 10)
        self.sn2 = neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), detach_
↪reset=True, backend=backend)

    def forward(self, x: torch.Tensor):
        # x.shape = [N, C, H, W]
        x.squeeze_(1)  # [N, H, W]
        x = x.permute(2, 0, 1)  # [W, N, H]
        x = self.fc1(x)
        x = self.sn1(x)
        x = self.sy1(x)
        x = self.fc2(x)
        x = self.sn2(x)
        return x.mean(0)
```

We add a recurrent connection *spikingjelly.clock_driven.layer.LinearRecurrentContainer* from the first spiking neurons layer's output to itself and get `FeedBackNet`:

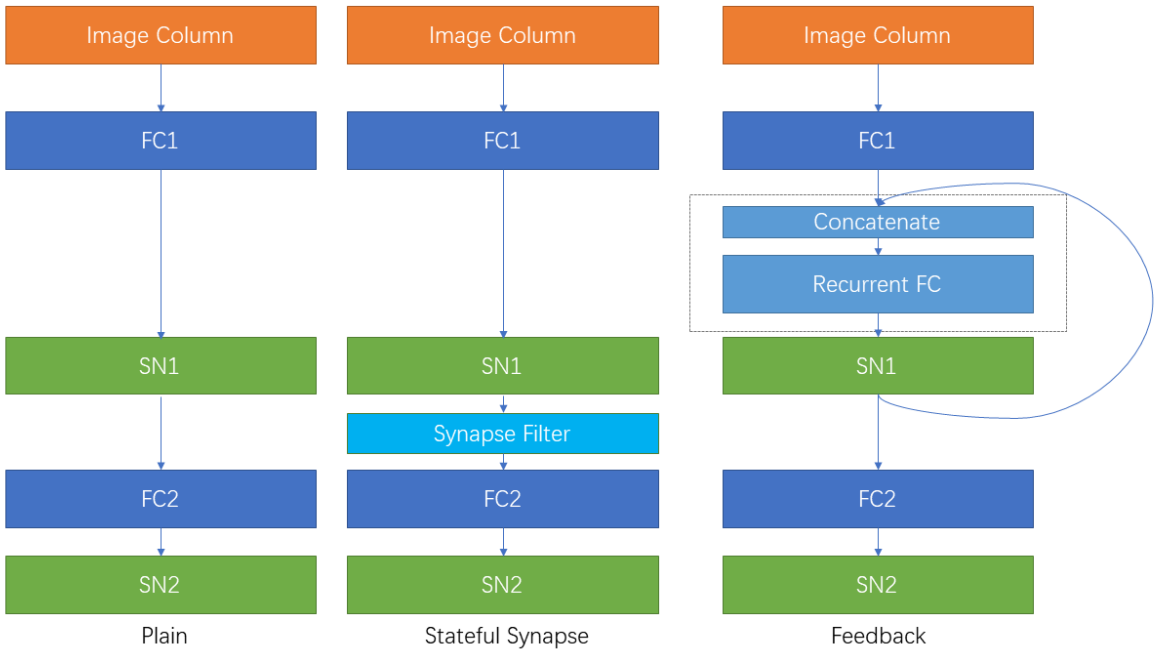
```
class FeedBackNet(nn.Module):
    def __init__(self):
        super().__init__()
```

(续下页)

(接上页)

```
        self.fc1 = nn.Linear(28, 32)
        self.sn1 = layer.MultiStepContainer(
            layer.LinearRecurrentContainer(
                neuron.IFNode(surrogate_function=surrogate.ATan(), detach_reset=True),
                32, 32
            )
        )
        self.fc2 = nn.Linear(32, 10)
        self.sn2 = neuron.MultiStepIFNode(surrogate_function=surrogate.ATan(), detach_
↪reset=True, backend=backend)

    def forward(self, x: torch.Tensor):
        # x.shape = [N, C, H, W]
        x.squeeze_(1)  # [N, H, W]
        x = x.permute(2, 0, 1)  # [W, N, H]
        x = seq_to_ann_forward(x, self.fc1)
        x = self.sn1(x)
        x = seq_to_ann_forward(x, self.fc2)
        x = self.sn2(x)
        return x.mean(0)
```

The following figure shows the three networks:



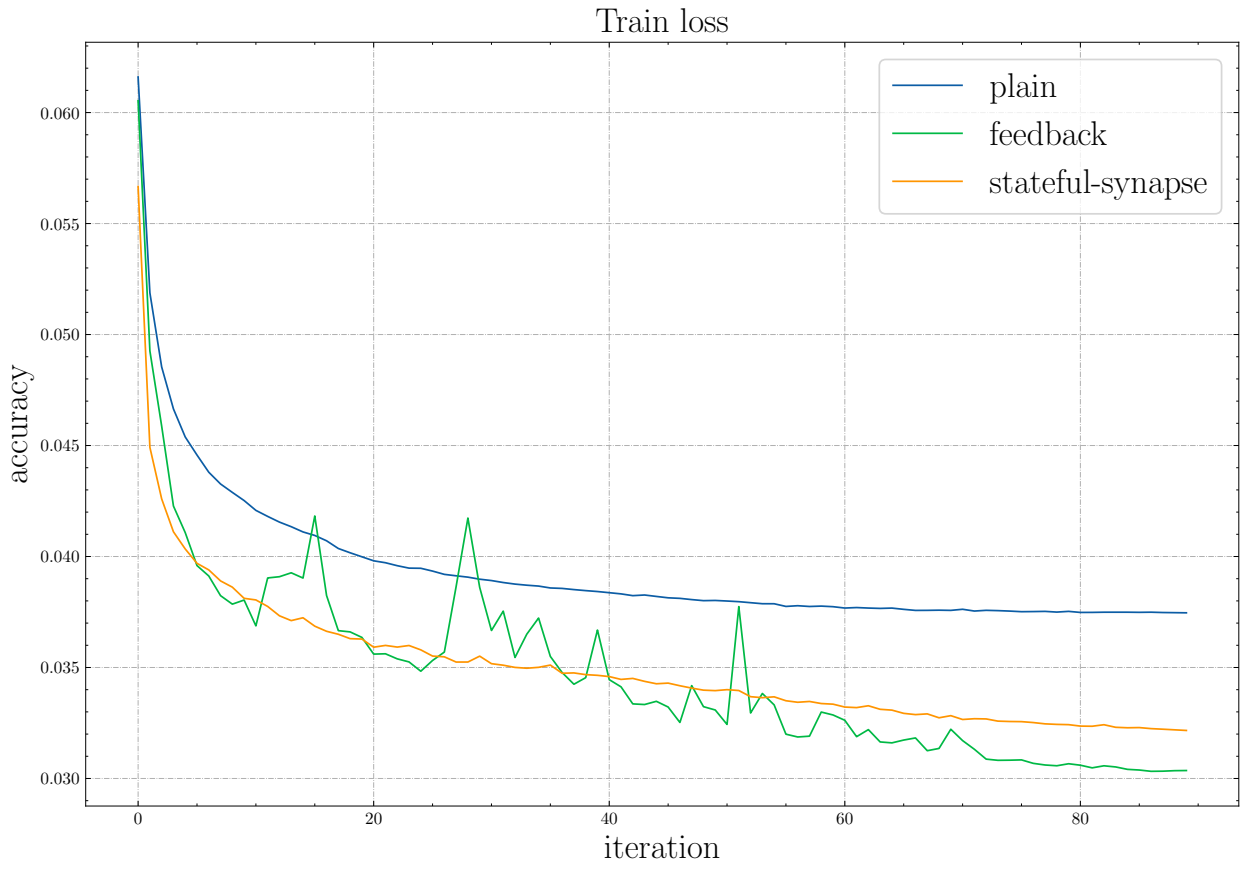
The complete codes are available at spikingjelly.clock_driven.examples.rsnn_sequential_fmnist. We can run it in console, and the running arguments are

```
(pytorch-env) PS C:/Users/fw> python -m spikingjelly.clock_driven.examples.rsnn_
↪sequential_fmnist --h
usage: rsnn_sequential_fmnist.py [-h] [--data-path DATA_PATH] [--device DEVICE] [-b␣
↪BATCH_SIZE] [--epochs N] [-j N]
                                 [--lr LR] [--opt OPT] [--lrs LRS] [--step-size STEP_
↪SIZE] [--step-gamma STEP_GAMMA]
                                 [--cosa-tmax COSA_TMAX] [--momentum M] [--wd W] [--
↪output-dir OUTPUT_DIR]
                                 [--resume RESUME] [--start-epoch N] [--cache-
↪dataset] [--amp] [--tb] [--model MODEL]

PyTorch Classification Training

optional arguments:
  -h, --help            show this help message and exit
  --data-path DATA_PATH
                        dataset
  --device DEVICE       device
  -b BATCH_SIZE, --batch-size BATCH_SIZE
  --epochs N            number of total epochs to run
  -j N, --workers N     number of data loading workers (default: 16)
  --lr LR               initial learning rate
  --opt OPT             optimizer (sgd or adam)
  --lrs LRS             lr schedule (cosa(CosineAnnealingLR), step(StepLR)) or None
  --step-size STEP_SIZE
                        step_size for StepLR
  --step-gamma STEP_GAMMA
                        gamma for StepLR
  --cosa-tmax COSA_TMAX
                        T_max for CosineAnnealingLR. If none, it will be set to epochs
  --momentum M          Momentum for SGD
  --wd W, --weight-decay W
                        weight decay (default: 0)
  --output-dir OUTPUT_DIR
                        path where to save
  --resume RESUME       resume from checkpoint
  --start-epoch N       start epoch
  --cache-dataset       Cache the datasets for quicker initialization. It also␣
↪serializes the transforms
  --amp                 Use AMP training
  --tb                  Use TensorBoard to record logs
  --model MODEL         "plain", "feedback", or "stateful-synapse"
```

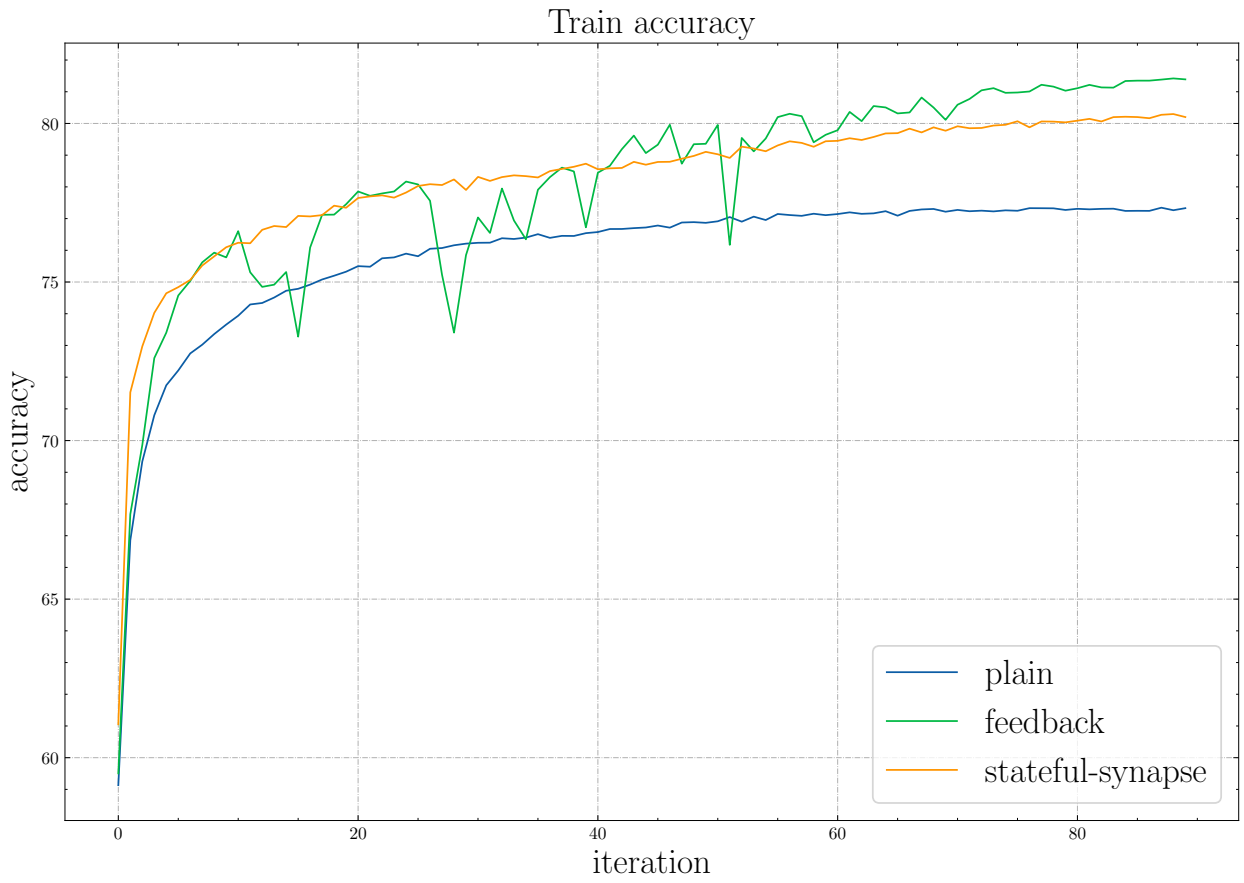
Let us train the three networks:

```
python -m spikingjelly.clock_driven.examples.rsnn_sequential_fmnist --data-path /raid/
↪wfang/datasets/FashionMNIST --tb --device cuda:0 --amp --model plain

python -m spikingjelly.clock_driven.examples.rsnn_sequential_fmnist --data-path /raid/
↪wfang/datasets/FashionMNIST --tb --device cuda:1 --amp --model feedback

python -m spikingjelly.clock_driven.examples.rsnn_sequential_fmnist --data-path /raid/
↪wfang/datasets/FashionMNIST --tb --device cuda:2 --amp --model stateful-synapse
```

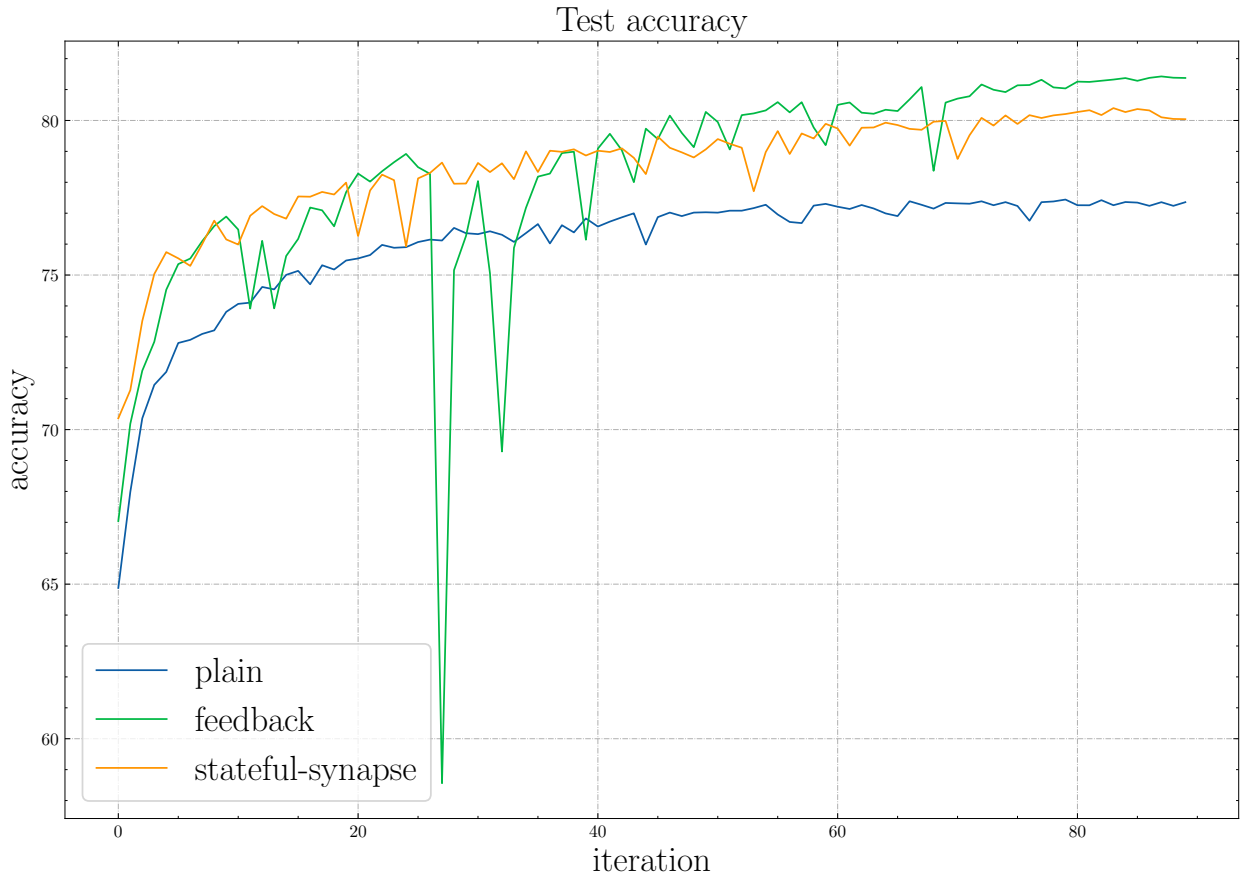
The train loss is:



The train accuracy is:



The test accuracy is:



We can find that both `feedback` and `stateful-synapse` have higher accuracy than `plain`, indicating that recurrent connections and stateful synapses can promote the network's ability to learn long-term dependencies.

7.1.16 Train Large-Scale SNN

Author: fangwei123456

Use networks from `spikingjelly.clock_driven.model`

spikingjelly.clock_driven.model provides some classic networks. We use *spikingjelly.clock_driven.model.spiking_resnet* as the example to show how to use them.

Most of networks in *spikingjelly.clock_driven.model* have two version: single-step and multi-step. We can create a single-step Spiking ResNet-18[1] like this:

```
import torch
import torch.nn.functional as F
```

(续下页)

[1] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

(接上页)

```
from spikingjelly.clock_driven import neuron, surrogate, functional
from spikingjelly.clock_driven.model import spiking_resnet

net = spiking_resnet.spiking_resnet18(pretrained=False, progress=True, single_step_
↪neuron=neuron.IFNode, v_threshold=1., surrogate_function=surrogate.ATan())
print(net)
```

As the arguments in `spiking_resnet18(pretrained=False, progress=True, single_step_neuron: callable=None, **kwargs)`, `single_step_neuron` is the single-step neuron, and `**kwargs` are args for the neuron. If we set `pretrained=True`, the Spiking ResNet-18 will load pre-trained parameters from ResNet-18 (ANN, rather than SNN). The outputs are:

```
SpikingResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),␣
↪bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
  (sn1): IFNode(
    v_threshold=1.0, v_reset=0.0, detach_reset=False
    (surrogate_function): ATan(alpha=2.0, spiking=True)
  )
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_
↪mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
    (1): BasicBlock(
```

(续下页)

(接上页)

```
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): BasicBlock(
```

(续下页)

(接上页)

```
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): BasicBlock(
```

(续下页)

(接上页)

```
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): BasicBlock(
```

(续下页)

(接上页)

```
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): IFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

The inputs of single-step network do not contain the time dimension. We need to give inputs at each time-step to the network:

```
net = spiking_resnet.spiking_resnet18(pretrained=False, progress=True, single_step_
↪neuron=neuron.IFNode, v_threshold=1., surrogate_function=surrogate.ATan())
T = 4
N = 2
x = torch.rand([T, N, 3, 224, 224])
fr = 0.
with torch.no_grad():
    for t in range(T):
        fr += net(x[t])
    fr /= T
print('firing rate =', fr)
```

To build a multi-step network, we should use *spikingjelly.clock_driven.model.spiking_resnet.multi_step_spiking_resnet18*, rather than *spikingjelly.clock_driven.model.spiking_resnet.spiking_resnet18*, and use the multi-step neuron:

```
net_ms = spiking_resnet.multi_step_spiking_resnet18(pretrained=False, progress=True,␣
↪multi_step_neuron=neuron.MultiStepIFNode, v_threshold=1., surrogate_
↪function=surrogate.ATan(), backend='torch')
```

(续下页)

(接上页)

```
print(net_ms)
```

The outputs are:

```
MultiStepSpikingResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),␣
↪bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
  (sn1): MultiStepIFNode(
    v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
    (surrogate_function): ATan(alpha=2.0, spiking=True)
  )
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_
↪mode=False)
  (layer1): Sequential(
    (0): MultiStepBasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
    (1): MultiStepBasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
```

(续下页)

(接上页)

```
↪bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (layer2): Sequential(
    (0): MultiStepBasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): MultiStepBasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
```

(续下页)

(接上页)

```
↪bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (layer3): Sequential(
    (0): MultiStepBasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): MultiStepBasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
```

(续下页)

(接上页)

```
↪bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (layer4): Sequential(
    (0): MultiStepBasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      )
    )
    (1): MultiStepBasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
↪bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn1): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),␣
```

(续下页)

(接上页)

```
↪bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
      (sn2): MultiStepIFNode(
        v_threshold=1.0, v_reset=0.0, detach_reset=False, backend=cupy
        (surrogate_function): ATan(alpha=2.0, spiking=True)
      )
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

The inputs for multi-step network should have the time dimension:

```
net = spiking_resnet.spiking_resnet18(pretrained=False, progress=True, single_step_
↪neuron=neuron.IFNode, v_threshold=1.,
                                      surrogate_function=surrogate.ATan())
T = 4
N = 2
x = torch.rand([T, N, 3, 224, 224])
fr = 0.
with torch.no_grad():
    for t in range(T):
        fr += net(x[t])
    fr /= T

net_ms = spiking_resnet.multi_step_spiking_resnet18(pretrained=False, progress=True,␣
↪multi_step_neuron=neuron.MultiStepIFNode, v_threshold=1., surrogate_
↪function=surrogate.ATan(), backend='torch')

net_ms.load_state_dict(net.state_dict())
with torch.no_grad():
    print('mse of single/multi step network outputs', F.mse_loss(net_ms(x).mean(0),␣
↪fr))
```

However, this network also supports for inputs without time dimension, as long as we set `T` when building or after building the network.

Setting `T` when building:

```
net_ms = spiking_resnet.multi_step_spiking_resnet18(pretrained=False, progress=True,␣
↪T=4, multi_step_neuron=neuron.MultiStepIFNode, v_threshold=1., surrogate_
↪function=surrogate.ATan(), backend='torch')
```

Or setting `T` after building:

```
net_ms = spiking_resnet.multi_step_spiking_resnet18(pretrained=False, progress=True,␣
↪multi_step_neuron=neuron.MultiStepIFNode, v_threshold=1., surrogate_
↪function=surrogate.ATan(), backend='torch')
net_ms.T = 4
```

The network will repeat inputs in time dimension, which is same with we do it manually:

```
net_ms = spiking_resnet.multi_step_spiking_resnet18(pretrained=False, progress=True,␣
↪multi_step_neuron=neuron.MultiStepIFNode, v_threshold=1., surrogate_
↪function=surrogate.ATan(), backend='torch')
T = 4
N = 2

with torch.no_grad():
    x = torch.rand([N, 3, 224, 224])
    y1 = net_ms(x.unsqueeze(0).repeat(T, 1, 1, 1, 1))
    functional.reset_net(net_ms)
    net_ms.T = T
    y2 = net_ms(x)
    print(F.mse_loss(y1, y2))
```

The outputs are:

```
tensor(0.)
```

However, it is more efficient to let network to repeat. Refer to *Clock driven: Use convolutional SNN to identify Fashion-MNIST* for the reason.

Training on ImageNet

ImageNet[2] is a popular baseline dataset for computer version, which is challenging for SNNs. SpikingJelly provides a code example to train on ImageNet, which is available at spikingjelly.clock_driven.model.train_imagenet .The example is written by referring torchvision .We can use it to train our network on ImageNet after we build the network, loss and how to calculate accuracy. Here is an example:

```
import torch
import torch.nn.functional as F
from spikingjelly.clock_driven.model import train_imagenet, spiking_resnet, train_
↪classify
from spikingjelly.clock_driven import neuron, surrogate
```

(续下页)

[2] Deng, Jia, et al. "Imagenet: A large-scale hierarchical image database." 2009 IEEE conference on computer vision and pattern recognition. IEEE, 2009.

(接上页)

```
def ce_loss(x_seq: torch.Tensor, label: torch.Tensor):
    # x_seq.shape = [T, N, C]
    return F.cross_entropy(input=x_seq.mean(0), target=label)

def cal_acc1_acc5(output, target):
    return train_classify.default_cal_acc1_acc5(output.mean(0), target)


if __name__ == '__main__':
    net = spiking_resnet.multi_step_spiking_resnet18(T=4, multi_step_neuron=neuron.
↪MultiStepIFNode, surrogate_function=surrogate.ATan(), detach_reset=True, backend=
↪'cupy')
    args = train_imagenet.parse_args()
    train_imagenet.main(model=net, criterion=ce_loss, args=args, cal_acc1_acc5=cal_
↪acc1_acc5)
```

Let us save these codes as *resnet18_imagenet.py*. The running arguments are:

```
(pytorch-env) wfang@onebrain-dgx-a100-01:~/ssd/temp_dir$ python resnet18_imagenet.py -
↪h

                            [--step-gamma STEP_GAMMA] [--cosa-tmax COSA_TMAX] [--
↪momentum M] [--wd W] [--output-dir OUTPUT_DIR] [--resume RESUME] [--start-epoch N]␣
↪[--cache-dataset]
                            [--sync-bn] [--amp] [--world-size WORLD_SIZE] [--dist-url␣
↪DIST_URL] [--tb] [--T T] [--local_rank LOCAL_RANK]

PyTorch Classification Training

optional arguments:
  -h, --help            show this help message and exit
  --data-path DATA_PATH
                        dataset
  --device DEVICE       device
  -b BATCH_SIZE, --batch-size BATCH_SIZE
  --epochs N            number of total epochs to run
  -j N, --workers N     number of data loading workers (default: 16)
  --lr LR               initial learning rate
  --opt OPT             optimizer (sgd or adam)
  --lrs LRS             lr schedule (cosa(CosineAnnealingLR), step(StepLR)) or None
  --step-size STEP_SIZE
                        step_size for StepLR
  --step-gamma STEP_GAMMA
```

(续下页)

(接上页)

```
                        gamma for StepLR
  --cosa-tmax COSA_TMAX
                        T_max for CosineAnnealingLR. If none, it will be set to epochs
  --momentum M          Momentum for SGD
  --wd W, --weight-decay W
                        weight decay (default: 0)
  --output-dir OUTPUT_DIR
                        path where to save
  --resume RESUME       resume from checkpoint
  --start-epoch N       start epoch
  --cache-dataset       Cache the datasets for quicker initialization. It also␣
↪serializes the transforms
  --sync-bn             Use sync batch norm
  --amp                 Use AMP training
  --world-size WORLD_SIZE
                        number of distributed processes
  --dist-url DIST_URL   url used to set up distributed training
  --tb                  Use TensorBoard to record logs
  --T T                 simulation steps
  --local_rank LOCAL_RANK
```

Training on a GPU:

```
python resnet18_imagenet.py --data-path /raid/wfang/datasets/ImageNet --lr 0.1 --opt␣
↪sgd --lrs cosa --amp --tb --device cuda:7
```

Training on multi-gpu:

```
python -m torch.distributed.launch --nproc_per_node=8 resnet18_imagenet.py --data-
↪path /raid/wfang/datasets/ImageNet --lr 0.1 --opt sgd --lrs cosa --amp --tb
```

7.2 Modules Docs

- *APIs*

7.3 Indices and tables

- Index
- Module Index
- Search Page

7.4 Citation

If you use SpikingJelly in your work, please cite it as follows:

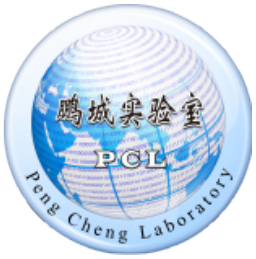
```
@misc{SpikingJelly,
    title = {SpikingJelly},
    author = {Fang, Wei and Chen, Yanqi and Ding, Jianhao and Chen, Ding and Yu,␣
↪Zhaofei and Zhou, Huihui and Tian, Yonghong and other contributors},
    year = {2020},
    howpublished = {\url{https://github.com/fangwei123456/spikingjelly}},
   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

Note: To specify the version of framework you are using, the default value YYYY-MM-DD in the note field should be replaced with the date of the last change of the framework you are using, i.e. the date of the latest commit.

Publications using SpikingJelly are recorded in Publications using SpikingJelly. If you use SpikingJelly in your paper, you can also add it to this table by pull request.

7.5 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.

7.5.1 spikingjelly.clock_driven package

spikingjelly.clock_driven.examples package

Subpackages

spikingjelly.clock_driven.examples.common package

Submodules

spikingjelly.clock_driven.examples.common.multiprocessing_env module

spikingjelly.clock_driven.examples.common.multiprocessing_env.**worker**(*remote*, *parent_remote*, *env_fn_wrapper*)

class spikingjelly.clock_driven.examples.common.multiprocessing_env.**VecEnv**(*num_envs*, *observation_space*, *action_space*)

基类：object

An abstract asynchronous, vectorized environment.

reset()

Reset all the environments and return an array of observations, or a tuple of observation arrays. If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

step_async(*actions*)

Tell all the environments to start taking a step with the given actions. Call step_wait() to get the results of the step. You should not call this if a step_async run is already pending.

step_wait()

Wait for the step taken with step_async(). Returns (obs, rews, dones, infos):

- **obs: an array of observations, or a tuple of**
  arrays of observations.
- rews: an array of rewards
- dones: an array of "episode done" booleans
- infos: a sequence of info objects

close()

Clean up the environments' resources.

step(*actions*)

class spikingjelly.clock_driven.examples.common.multiprocessing_env.**CloudpickleWrapper**(*x*)

基类: object

Uses cloudpickle to serialize contents (otherwise multiprocessing tries to use pickle)

class spikingjelly.clock_driven.examples.common.multiprocessing_env.**SubprocVecEnv**(*env_fns*, *spaces=None*

基类: *VecEnv*

envs: list of gym environments to run in subprocesses

step_async(*actions*)

step_wait()

reset()

reset_task()

close()

Module contents

Submodules

spikingjelly.clock_driven.examples.A2C module

spikingjelly.clock_driven.examples.DQN_state module

class spikingjelly.clock_driven.examples.DQN_state.**ReplayMemory**(*capacity*)

基类：object

push(**args*)

Saves a transition.

sample(*batch_size*)

class spikingjelly.clock_driven.examples.DQN_state.**DQN**(*input_size*, *hidden_size*, *output_size*)

基类：Module

forward(*x*)

training: bool

spikingjelly.clock_driven.examples.PPO module

spikingjelly.clock_driven.examples.PPO.**make_env**()

class spikingjelly.clock_driven.examples.PPO.**ActorCritic**(*num_inputs*, *num_outputs*, *hidden_size*, *std=0.0*)

基类：Module

forward(*x*)

training: bool

spikingjelly.clock_driven.examples.Spiking_A2C module

class spikingjelly.clock_driven.examples.Spiking_A2C.**NonSpikingLIFNode**(**args*, ***kwargs*)

基类：*LIFNode*

forward(*dv: Tensor*)

training: bool

class spikingjelly.clock_driven.examples.Spiking_A2C.**ActorCritic**(*num_inputs*, *num_outputs*, *hidden_size*, *T=16*)

基类：Module

forward(*x*)

training: bool

spikingjelly.clock_driven.examples.Spiking_DQN_state module

class spikingjelly.clock_driven.examples.Spiking_DQN_state.**Transition**(*state*, *action*, *next_state*, *reward*)

基类：tuple

Create new instance of Transition(state, action, next_state, reward)

property action

Alias for field number 1

property next_state

Alias for field number 2

property reward

Alias for field number 3

property state

Alias for field number 0

class spikingjelly.clock_driven.examples.Spiking_DQN_state.**ReplayMemory**(*capacity*)

基类：object

push(**args*)

sample(*batch_size*)

class spikingjelly.clock_driven.examples.Spiking_DQN_state.**NonSpikingLIFNode**(**args*, ***kwargs*)

基类：*LIFNode*

forward(*dv: Tensor*)

training: bool

class spikingjelly.clock_driven.examples.Spiking_DQN_state.**DQSN**(*input_size*, *hidden_size*, *output_size*, *T=16*)

基类：Module

forward(*x*)

training: bool

spikingjelly.clock_driven.examples.Spiking_DQN_state.**train**(*use_cuda*, *model_dir*, *log_dir*, *env_name*, *hidden_size*, *num_episodes*, *seed*)

spikingjelly.clock_driven.examples.Spiking_DQN_state.**play**(*use_cuda*, *pt_path*, *env_name*, *hidden_size*, *played_frames=60*, *save_fig_num=0*, *fig_dir=None*, *figsize=(12, 6)*, *firing_rates_plot_type='bar'*, *heatmap_shape=None*)

spikingjelly.clock_driven.examples.Spiking_PPO module

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.**relu**

基类：Function

static forward(*ctx*, *x*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation.**Ba**

基类：Module

spiking()

forward(*dv: Tensor*)

reset()

training: bool

class spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation.**Li**

基类：*BaseNode*

forward(*dv: Tensor*)

training: bool

class spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation.**IF**

基类：*BaseNode*

forward(*dv: Tensor*)

training: bool

class spikingjelly.clock_driven.examples.
cifar10_r11_enabling_spikebased_backpropagation.**ResNet11**

基类：Module

forward(*x*)

reset_()

training: bool

spikingjelly.clock_driven.examples.cifar10_r11_enabling_spikebased_backpropagation.**main**()

spikingjelly.clock_driven.examples.classify_dvsg module

class spikingjelly.clock_driven.examples.classify_dvsg.**VotingLayer**(*voter_num: int*)

基类：Module

forward(*x: Tensor*)

training: bool

class spikingjelly.clock_driven.examples.classify_dvsg.**PythonNet**(*channels: int*)

基类：Module

forward(*x: Tensor*)

static conv3x3(*in_channels: int*, *out_channels*)

training: bool

spikingjelly.clock_driven.examples.classify_dvsg.**main**()

- *API in English*

用于分类 DVS128 Gesture 数据集的代码样例。网络结构来自于 Incorporating Learnable Membrane Time Constant to Enhance Learning of Spiking Neural Networks。

```
usage: classify_dvsg.py [-h] [-T T] [-device DEVICE] [-b B] [-epochs N] [-j N] [-
↪channels CHANNELS] [-data_dir DATA_DIR] [-out_dir OUT_DIR] [-resume RESUME] [-
↪amp] [-cupy] [-opt OPT] [-lr LR] [-momentum MOMENTUM] [-lr_scheduler LR_
↪SCHEDULER] [-step_size STEP_SIZE] [-gamma GAMMA] [-T_max T_MAX]

Classify DVS128 Gesture

optional arguments:
  -h, --help            show this help message and exit
  -T T                  simulating time-steps
  -device DEVICE        device
  -b B                  batch size
  -epochs N             number of total epochs to run
  -j N                  number of data loading workers (default: 4)
  -channels CHANNELS    channels of Conv2d in SNN
  -data_dir DATA_DIR    root dir of DVS128 Gesture dataset
  -out_dir OUT_DIR      root dir for saving logs and checkpoint
  -resume RESUME        resume from the checkpoint path
  -amp                  automatic mixed precision training
  -cupy                 use CUDA neuron and multi-step forward mode
  -opt OPT              use which optimizer. SDG or Adam
  -lr LR                learning rate
```

(续下页)

(接上页)

```
  -momentum MOMENTUM    momentum for SGD
  -lr_scheduler LR_SCHEDULER
                        use which schedule. StepLR or CosALR
  -step_size STEP_SIZE  step_size for StepLR
  -gamma GAMMA          gamma for StepLR
  -T_max T_MAX          T_max for CosineAnnealingLR
```

运行示例：

```
python -m spikingjelly.clock_driven.examples.classify_dvsg -data_dir /userhome/
↪datasets/DVS128Gesture -out_dir ./logs -amp -opt Adam -device cuda:0 -lr_
↪scheduler CosALR -T_max 64 -cupy -epochs 1024
```

阅读教程分类 *DVS128 Gesture* 以获得更多信息。

- 中文 *API*

The code example for classifying the DVS128 Gesture dataset. The network structure is from Incorporating Learnable Membrane Time Constant to Enhance Learning of Spiking Neural Networks.

```
usage: classify_dvsg.py [-h] [-T T] [-device DEVICE] [-b B] [-epochs N] [-j N] [-
↪channels CHANNELS] [-data_dir DATA_DIR] [-out_dir OUT_DIR] [-resume RESUME] [-
↪amp] [-cupy] [-opt OPT] [-lr LR] [-momentum MOMENTUM] [-lr_scheduler LR_
↪SCHEDULER] [-step_size STEP_SIZE] [-gamma GAMMA] [-T_max T_MAX]

Classify DVS128 Gesture

optional arguments:
  -h, --help            show this help message and exit
  -T T                  simulating time-steps
  -device DEVICE        device
  -b B                  batch size
  -epochs N             number of total epochs to run
  -j N                  number of data loading workers (default: 4)
  -channels CHANNELS    channels of Conv2d in SNN
  -data_dir DATA_DIR    root dir of DVS128 Gesture dataset
  -out_dir OUT_DIR      root dir for saving logs and checkpoint
  -resume RESUME        resume from the checkpoint path
  -amp                  automatic mixed precision training
  -cupy                 use CUDA neuron and multi-step forward mode
  -opt OPT              use which optimizer. SDG or Adam
  -lr LR                learning rate
  -momentum MOMENTUM    momentum for SGD
  -lr_scheduler LR_SCHEDULER
                        use which schedule. StepLR or CosALR
```

(续下页)

(接上页)

```
  -step_size STEP_SIZE  step_size for StepLR
  -gamma GAMMA          gamma for StepLR
  -T_max T_MAX          T_max for CosineAnnealingLR
```

Running Example:

```
python -m spikingjelly.clock_driven.examples.classify_dvsg -data_dir /userhome/
↪datasets/DVS128Gesture -out_dir ./logs -amp -opt Adam -device cuda:0 -lr_
↪scheduler CosALR -T_max 64 -cupy -epochs 1024
```

See the tutorial *Classify DVS128 Gesture* for more details.

spikingjelly.clock_driven.examples.conv_fashion_mnist module

class spikingjelly.clock_driven.examples.conv_fashion_mnist.**PythonNet**(*T*)

基类：Module

forward(*x*)

training: bool

class spikingjelly.clock_driven.examples.conv_fashion_mnist.**CupyNet**(*T*)

基类：Module

forward(*x*)

training: bool

spikingjelly.clock_driven.examples.conv_fashion_mnist.**main**()

- *API in English*

Classify Fashion-MNIST

optional arguments:

-h, --help	show this help message and exit
-T T	simulating time-steps

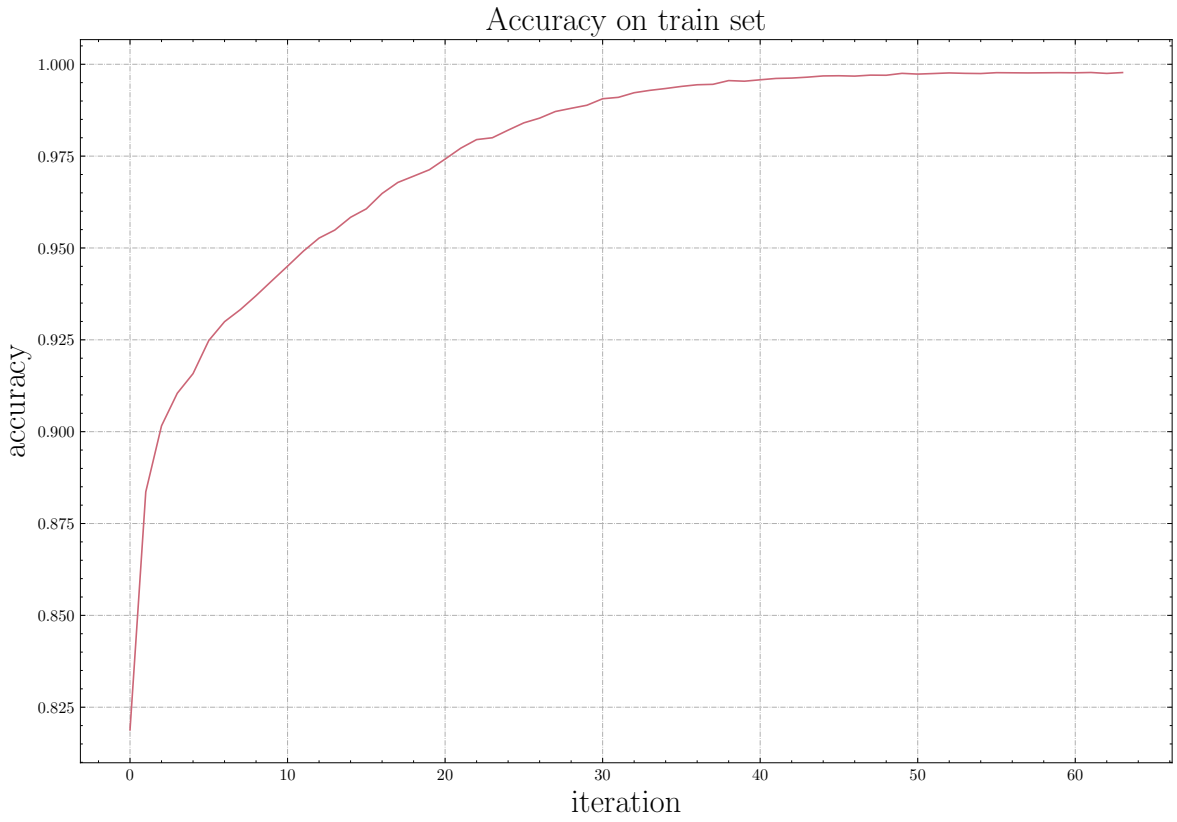
-device DEVICE device -b B batch size -epochs N number of total epochs to run -j N number of data loading workers (default: 4) -data_dir DATA_DIR root dir of Fashion-MNIST dataset -out_dir OUT_DIR root dir for saving logs and checkpoint -resume RESUME resume from the checkpoint path -amp automatic mixed precision training -cupy use cupy neuron and multi-step forward mode -opt OPT use which optimizer. SDG or Adam -lr LR learning rate -momentum MOMENTUM momentum for SGD -lr_scheduler LR_SCHEDULER

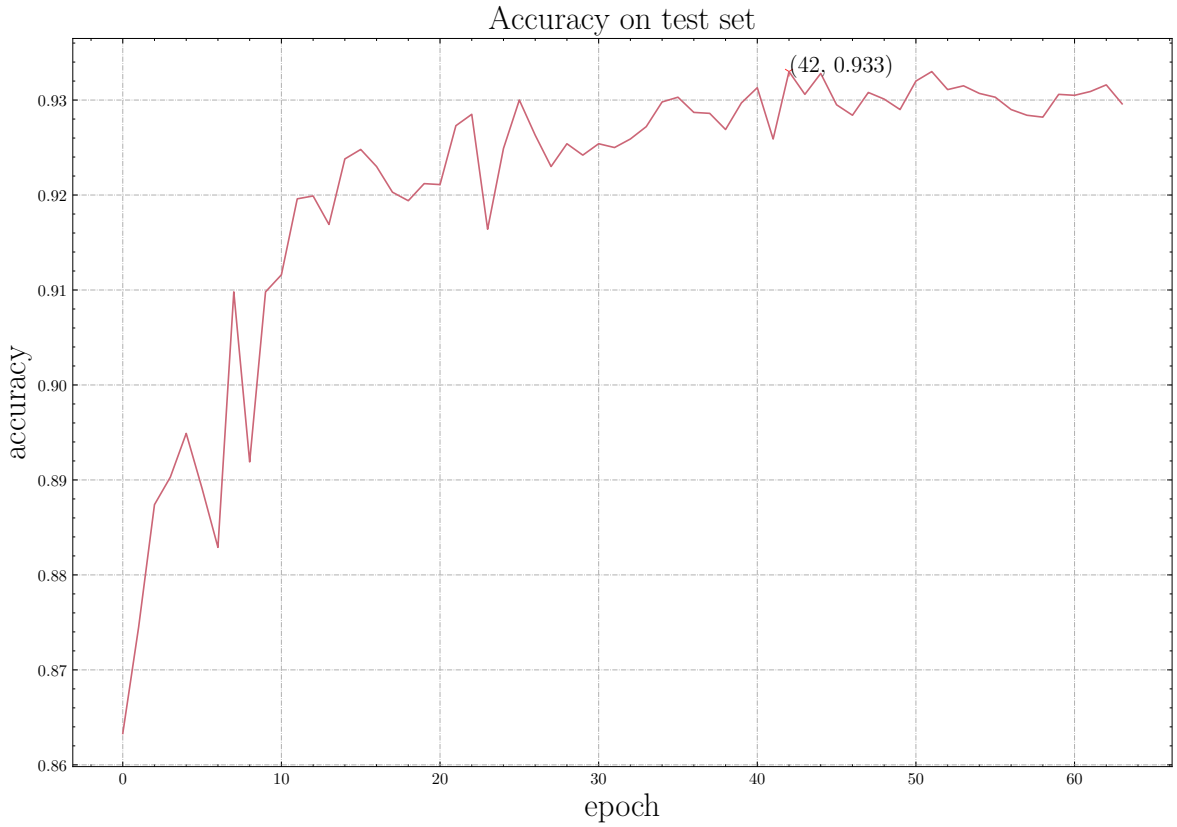
use which schedule. StepLR or CosALR

-step_size STEP_SIZE step_size for StepLR -gamma GAMMA gamma for StepLR -T_max T_MAX T_max for CosineAnnealingLR

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：



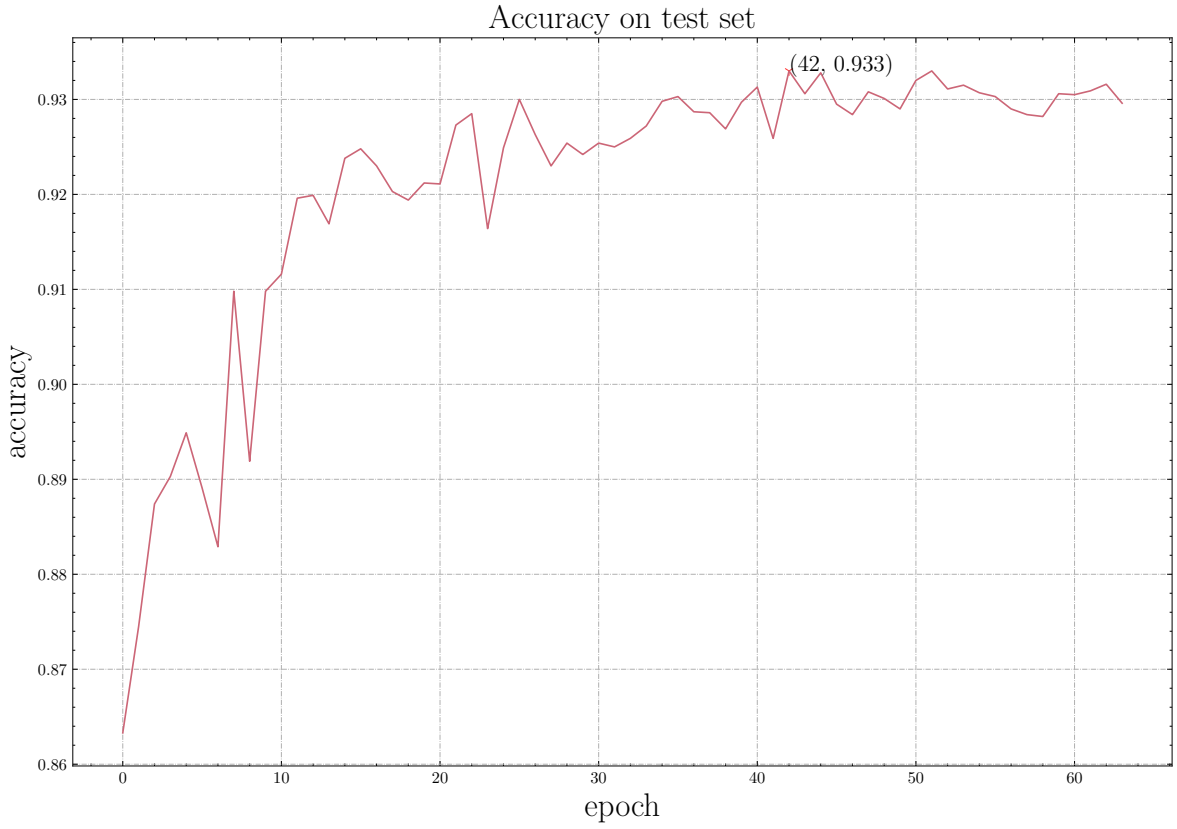


- 中文 *API*

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:


Accuracy on train set
accuracy
iteration
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0
10
20
30
40
50
60


Accuracy on test set
(42, 0.933)
accuracy
epoch
0.93
0.92
0.91
0.90
0.89
0.88
0.87
0.86
0
10
20
30
40
50
60

spikingjelly.clock_driven.examples.conv_fashion_mnist_cuda_lbl module

spikingjelly.clock_driven.examples.dqn_cart_pole module

class spikingjelly.clock_driven.examples.dqn_cart_pole.**Transition**(*state*, *action*, *next_state*, *reward*)

基类：tuple

Create new instance of Transition(state, action, next_state, reward)

property action

Alias for field number 1

property next_state

Alias for field number 2

property reward

Alias for field number 3

property state

Alias for field number 0

class spikingjelly.clock_driven.examples.dqn_cart_pole.**ReplayMemory**(*capacity*)

基类：object

push(**args*)

sample(*batch_size*)

class spikingjelly.clock_driven.examples.dqn_cart_pole.**NonSpikingLIFNode**(**args*, ***kwargs*)

基类：*LIFNode*

forward(*dv: Tensor*)

training: bool

class spikingjelly.clock_driven.examples.dqn_cart_pole.**DQSN**(*hidden_num*)

基类：Module

forward(*x*)

training: bool

spikingjelly.clock_driven.examples.dqn_cart_pole.**train**(*device*, *root*, *hidden_num=128*, *num_episodes=256*)

spikingjelly.clock_driven.examples.dqn_cart_pole.**play**(*device*, *pt_path*, *hidden_num*, *played_frames=60*, *save_fig_num=0*, *fig_dir=None*, *figsize=(12, 6)*, *firing_rates_plot_type='bar'*, *heatmap_shape=None*)

spikingjelly.clock_driven.examples.lif_fc_mnist module

spikingjelly.clock_driven.examples.lif_fc_mnist.**main**()

返回

None

- *API in English*

使用全连接-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 structure for classifying MNIST.

This function initials the network, starts trainingand shows accuracy on test dataset.

spikingjelly.clock_driven.examples.spiking_lstm_sequential_mnist module

class spikingjelly.clock_driven.examples.spiking_lstm_sequential_mnist.**Net**

基类：Module

forward(*x*)

training: **bool**

spikingjelly.clock_driven.examples.spiking_lstm_sequential_mnist.**main**()

spikingjelly.clock_driven.examples.spiking_lstm_text module

spikingjelly.clock_driven.examples.speechcommands module

Module contents

spikingjelly.clock_driven.encoding package

Module contents

class spikingjelly.clock_driven.encoding.**StatelessEncoder**

基类：Module

- *API in English*

无状态编码器的基类。无状态编码器 `encoder = StatelessEncoder()`，直接调用 `encoder(x)` 即可将 `x` 编码为 `spike`。

- *中文 API*

The base class of stateless encoder. The stateless encoder `encoder = StatelessEncoder()` can encode `x` to `spike` by `encoder(x)`.

abstract forward(*x: Tensor*)

- *API in English*

参数

x (*torch.Tensor*) –输入数据

返回

`spike`, shape 与 `x.shape` 相同

返回类型

torch.Tensor

- *中文 API*

参数

x (*torch.Tensor*) –input data

返回

`spike`, whose shape is same with `x.shape`

返回类型

torch.Tensor

training: bool

class spikingjelly.clock_driven.encoding.**StatefulEncoder**(*T: int*)

基类：MemoryModule

- *API in English*

参数

T (*int*) –编码周期。通常情况下，与 SNN 的仿真周期（总步长一致）

有状态编码器的基类。有状态编码器 `encoder = StatefulEncoder(T)`，编码器会在首次调用 `encoder(x)` 时对 `x` 进行编码。在第 ``t 次调用 `encoder(x)` 时会输出 `spike[t % T]`

```
encoder = StatefulEncoder(T)
s_list = []
for t in range(T):
    s_list.append(encoder(x))  # s_list[t] == spike[t]
```

- 中文 API

参数

T (*int*) –the encoding period. It is usually same with the total simulation time-steps of SNN

The base class of stateful encoder. The stateful encoder `encoder = StatefulEncoder(T)` will encode `x` to `spike` at the first time of calling `encoder(x)`. It will output `spike[t % T]` at the `t` -th calling

```
encoder = StatefulEncoder(T)
s_list = []
for t in range(T):
    s_list.append(encoder(x))  # s_list[t] == spike[t]
```

forward(*x: Optional[Tensor] = None*)

- API in English

参数

x (*torch.Tensor*) –输入数据

返回

`spike`, shape 与 `x.shape` 相同

返回类型

torch.Tensor

- 中文 API

参数

x (*torch.Tensor*) –input data

返回

`spike`, whose shape is same with `x.shape`

返回类型

torch.Tensor

abstract encode(*x: Tensor*)

- *API in English*

参数
x (*torch.Tensor*) –输入数据

返回
spike, shape 与 x.shape 相同

返回类型
torch.Tensor

- *中文 API*

参数
x (*torch.Tensor*) –input data

返回
spike, whose shape is same with x.shape

返回类型
torch.Tensor

extra_repr() → str

training: bool

class spikingjelly.clock_driven.encoding.**PeriodicEncoder**(*spike: Tensor*)

基类：*StatefulEncoder*

- *API in English*

参数
spike (*torch.Tensor*) –输入脉冲

周期性编码器，在第 t 次调用时输出 spike[t % T]，其中 T = spike.shape[0]

- *中文 API*

参数
spike (*torch.Tensor*) –the input spike

The periodic encoder that outputs spike[t % T] at t -th calling, where T = spike.shape[0]

encode(*spike: Tensor*)

training: bool

class spikingjelly.clock_driven.encoding.**LatencyEncoder**(*T: int*, *enc_function='linear'*)

基类：*StatefulEncoder*

- *API in English*

参数

- **T** (*int*) –最大（最晚）脉冲发放时刻
- **enc_function** (*str*) –定义使用哪个函数将输入强度转化为脉冲发放时刻，可以为 *linear* 或 *log*

延迟编码器，将 `0 <= x <= 1` 的输入转化为在 `0 <= t_f <= T-1` 时刻发放的脉冲。输入的强度越大，发放越早。当 `enc_function == 'linear'`

$$t_f(x) = (T-1)(1-x)$$

当 **enc_function == 'log'**

$$t_f(x) = (T-1) - ln(\alpha * x + 1)$$

其中 $lpha$ 满足 $t_f(1) = T - 1$

实例代码：

```
x = torch.rand(size=[8, 2])
print('x', x)
T = 20
encoder = LatencyEncoder(T)
for t range(T):
    print(encoder(x))
```

警告： 必须确保 `0 <= x <= 1`。

- 中文 *API*

参数

- **T** (*int*) –the maximum (latest) firing time
- **enc_function** (*str*) –how to convert intensity to firing time. *linear* or *log*

The latency encoder will encode `0 <= x <= 1` to spike whose firing time is `0 <= t_f <= T-1`. A larger `x` will cause a earlier firing time.

If `enc_function == 'linear'`

$$t_f(x) = (T-1)(1-x)$$

If `enc_function == 'log'`

$$t_f(x) = (T-1) - ln(\alpha * x + 1)$$

where $lpha$ satisfies $t_f(1) = T - 1$

Example: .. code-block:: python

x = torch.rand(size=[8, 2]) print(‘x’, x) T = 20 encoder = LatencyEncoder(T) for t range(T):

print(encoder(x))

Warning

The user must assert `0 <= x <= 1`.

`encode`(*x: Tensor*)

`training: bool`

`class` `spikingjelly.clock_driven.encoding.`**`PoissonEncoder`**

基类：*`StatelessEncoder`*

- *API in English*

无状态的泊松编码器。输出脉冲的发放概率与输入 `x` 相同。

> **警告：** 必须确保 `0 <= x <= 1`。

- *中文 API*

The poisson encoder will output spike whose firing probability is `x`。

Warning

The user must assert `0 <= x <= 1`.

`forward`(*x: Tensor*)

`training: bool`

class spikingjelly.clock_driven.encoding.**WeightedPhaseEncoder**(*K: int*)

基类：*StatefulEncoder*

- *API in English*

参数

K (*int*) –编码周期。通常情况下，与 SNN 的仿真周期（总步长一致）

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

- 中文 *API*

参数

K (*int*) –the encoding period. It is usually same with the total simulation time-steps of SNN

The weighted phase encoder, which is based on binary system. It will flatten `x` as a binary number. When `T=k`, it can encode $x \in [0, 1-2^{-K}]$ to different spikes. Here is the example from the origin paper:

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

encode(*x: Tensor*)

training: bool

spikingjelly.clock_driven.functional package

Module contents

spikingjelly.clock_driven.functional.**reset_net**(*net: 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.**spike_cluster**(*v: 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()
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: Tensor*, *labels: 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: Tensor*, *y: 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:* BaseNode, *label_one_hot: 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: 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: 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)
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.],
```

(续下页)

(接上页)

```
 [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.**multi_step_forward**(*x_seq: Tensor*, *single_step_module: Module*)

参数

- **x_seq** (*Tensor*) –shape=[T, batch_size, …]
- **single_step_module** (*torch.nn.Module or list or tuple or torch.nn.Sequential*) –a single-step module, or a list/tuple that contains single-step modules

返回

y_seq, shape=[T, batch_size, …]

返回类型

Tensor

See *spikingjelly.clock_driven.layer.MultiStepContainer* for more details.

spikingjelly.clock_driven.functional.**seq_to_ann_forward**(*x_seq: Tensor*, *stateless_module: Module*)

参数

- **x_seq** (*Tensor*) –shape=[T, batch_size, …]
- **stateless_module** (*torch.nn.Module or list or tuple or torch.nn.Sequential*) –a stateless module, e.g., ‘torch.nn.Conv2d’ or a list contains stateless modules, e.g., ‘[torch.nn.Conv2d, torch.nn.BatchNorm2d]

返回

y_seq, shape=[T, batch_size, …]

返回类型

Tensor

See *spikingjelly.clock_driven.layer.SeqToANNContainer* for more details.

spikingjelly.clock_driven.functional.**fused_conv2d_weight_of_convbn2d**(*conv2d: Conv2d*, *bn2d: BatchNorm2d*)

参数

- **conv2d** (*torch.nn.Conv2d*) –a Conv2d layer
- **bn2d** (*torch.nn.BatchNorm2d*) –a BatchNorm2d layer

返回

the weight of this fused module

返回类型

Tensor

A {Conv2d-BatchNorm2d} can be fused to a {Conv2d} module with BatchNorm2d's parameters being absorbed into Conv2d. This function returns the weight of this fused module.

Note

We assert *conv2d.bias* is *None*. See Disable bias for convolutions directly followed by a batch norm for more details.

---

spikingjelly.clock_driven.functional.**fused_conv2d_bias_of_convbn2d**(*conv2d: Conv2d*, *bn2d: BatchNorm2d*)

参数

- **conv2d** (*torch.nn.Conv2d*) –a Conv2d layer
- **bn2d** (*torch.nn.BatchNorm2d*) –a BatchNorm2d layer

返回

the bias of this fused module

返回类型

Tensor

A {Conv2d-BatchNorm2d} can be fused to a {Conv2d} module with BatchNorm2d's parameters being absorbed into Conv2d. This function returns the bias of this fused module.

Note

We assert *conv2d.bias* is *None*. See Disable bias for convolutions directly followed by a batch norm for more details.

---

spikingjelly.clock_driven.functional.**scale_fused_conv2d_weight_of_convbn2d**(*conv2d: Conv2d*, *bn2d: BatchNorm2d*, *k=None*, *b=None*)

参数

- **conv2d** (*torch.nn.Conv2d*) –a Conv2d layer
- **bn2d** (*torch.nn.BatchNorm2d*) –a BatchNorm2d layer

A {Conv2d-BatchNorm2d} can be fused to a {Conv2d} module with BatchNorm2d's parameters being absorbed into Conv2d. This function sets the weight of this fused module to *weight* * *k* + *b*.

Note

We assert *conv2d.bias* is *None*. See Disable bias for convolutions directly followed by a batch norm for more details.

---

spikingjelly.clock_driven.functional.**scale_fused_conv2d_bias_of_convbn2d**(*conv2d: Conv2d*, *bn2d: BatchNorm2d*, *k=None*, *b=None*)

参数

- **conv2d** (*torch.nn.Conv2d*) –a Conv2d layer
- **bn2d** (*torch.nn.BatchNorm2d*) –a BatchNorm2d layer

A {Conv2d-BatchNorm2d} can be fused to a {Conv2d} module with BatchNorm2d's parameters being absorbed into Conv2d. This function sets the bias of this fused module to *bias* * *k* + *b*.

Note

We assert *conv2d.bias* is *None*. See Disable bias for convolutions directly followed by a batch norm for more details.

---

spikingjelly.clock_driven.functional.**fuse_convbn2d**(*conv2d: Conv2d*, *bn2d: BatchNorm2d*, *k=None*, *b=None*)

参数

- **conv2d** (*torch.nn.Conv2d*) –a Conv2d layer
- **bn2d** (*torch.nn.BatchNorm2d*) –a BatchNorm2d layer

返回

the fused Conv2d layer

返回类型

torch.nn.Conv2d

A {Conv2d-BatchNorm2d} can be fused to a {Conv2d} module with BatchNorm2d's parameters being absorbed into Conv2d. This function returns the fused module.

Note

We assert *conv2d.bias* is *None*. See Disable bias for convolutions directly followed by a batch norm for more details.

---

spikingjelly.clock_driven.functional.**temporal_efficient_training_cross_entropy**(*x_seq: Tensor*, *target: LongTensor*)

参数

- **x_seq** (*Tensor*) –shape=[T, N, C, *], where C is the number of classes
- **target** (*torch.LongTensor*) –shape=[N], where 0 <= target[i] <= C-1

返回

the temporal efficient training cross entropy

返回类型

Tensor

The temporal efficient training (TET) cross entropy, which is the mean of cross entropy of each time-step.

Codes example:

```
def tet_ce_for_loop_version(x_seq: Tensor, target: torch.LongTensor):
    loss = 0.
    for t in range(x_seq.shape[0]):
        loss += F.cross_entropy(x_seq[t], target)
    return loss / x_seq.shape[0]

T = 8
```

(续下页)

(接上页)

```
N = 4
C = 10
x_seq = torch.rand([T, N, C])
target = torch.randint(low=0, high=C-1, size=[N])
print(tet_ce_for_loop_version(x_seq, target))
print(temporal_efficient_training_cross_entropy(x_seq, target))
```

Tip

The TET cross entropy is proposed by Temporal Efficient Training of Spiking Neural Network via Gradient Reweighting.

---

spikingjelly.clock_driven.functional.**kaiming_normal_conv_linear_weight**(*net: Module*)

- API in English

参数

net –任何属于 nn.Module 子类的网络

返回

None

使用 kaiming normal 初始化 *net* 中的所有 torch.nn._ConvNd 和 *:class:'torch.nn.Linear* 的权重（不包括偏置项）。参见 torch.nn.init.kaiming_normal_。

- 中文 API

参数

net –Any network inherits from nn.Module

返回

None

initialize all weights (not including bias) of torch.nn._ConvNd and torch.nn.Linear in *net* by the kaiming normal. See torch.nn.init.kaiming_normal_ for more details.

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：MemoryModule

- *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: Tensor*)

extra_repr() → str

training: bool

class spikingjelly.clock_driven.layer.**DCT**(*kernel_size*)

基类：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: Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**AXAT**(*in_features*, *out_features*)

基类：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: Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**Dropout**(*p=0.5*)

基类：MemoryModule

- *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: Tensor*)

forward(*x: Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**Dropout2d**(*p=0.2*)

基类：*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: Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**MultiStepDropout**(*p=0.5*)

基类：*Dropout*

- *API in English*

参数

p (*float*) –每个元素被设置为 0 的概率

spikingjelly.clock_driven.layer.Dropout 的多步版本。

小技巧： 阅读传播模式 以获取更多关于单步和多步传播的信息。

- 中文 *API*

参数

p (*float*) –probability of an element to be zeroed

The multi-step version of *spikingjelly.clock_driven.layer.Dropout*.

Tip

Read *Propagation Pattern* for more details about single-step and multi-step propagation.

---

forward(*x_seq: Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**MultiStepDropout2d**(*p=0.5*)

基类：*Dropout2d*

- *API in English*

参数

p (*float*) –每个元素被设置为 0 的概率

spikingjelly.clock_driven.layer.Dropout2d 的多步版本。

小技巧： 阅读传播模式 以获取更多关于单步和多步传播的信息。

- 中文 *API*

参数

p (*float*) –probability of an element to be zeroed

The multi-step version of *spikingjelly.clock_driven.layer.Dropout2d*.

Tip

Read *Propagation Pattern* for more details about single-step and multi-step propagation.

---

forward(*x_seq: Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**SynapseFilter**(*tau=100.0*, *learnable=False*)

基类：MemoryModule

- *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：

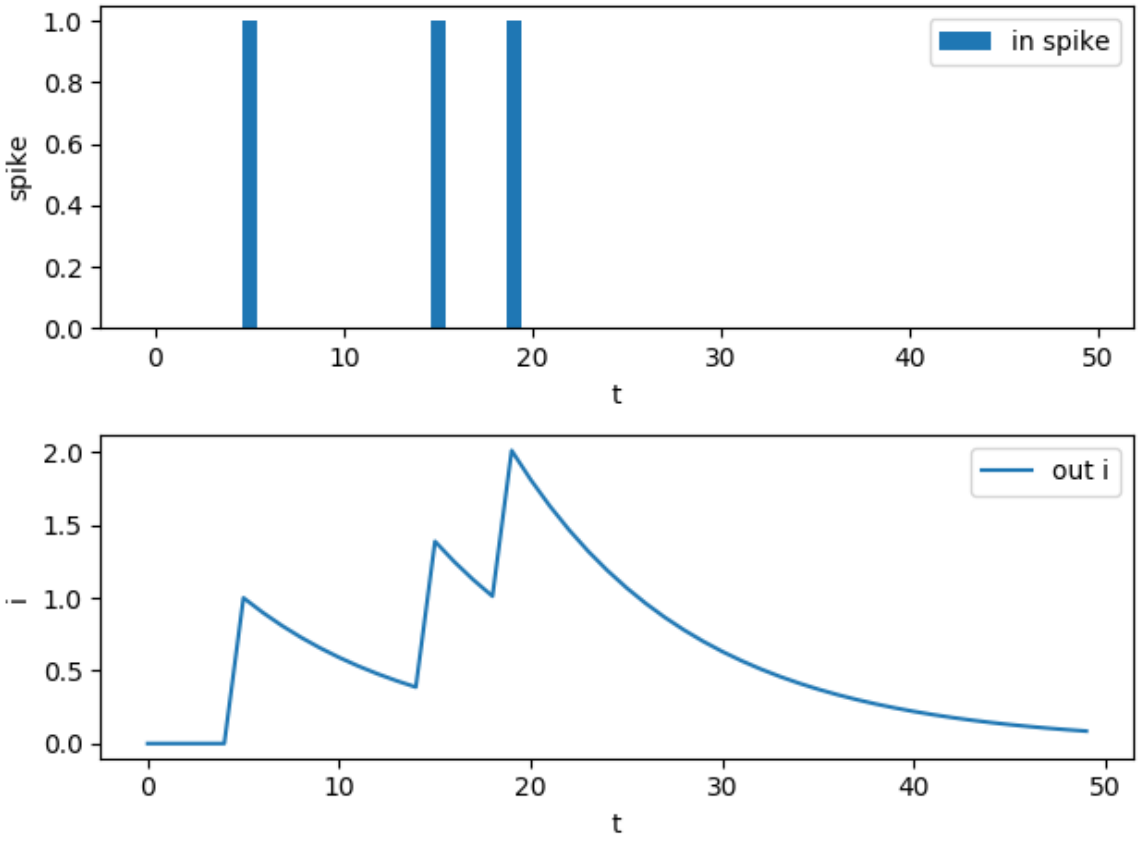
$$I(t) = I(t) + 1$$

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:

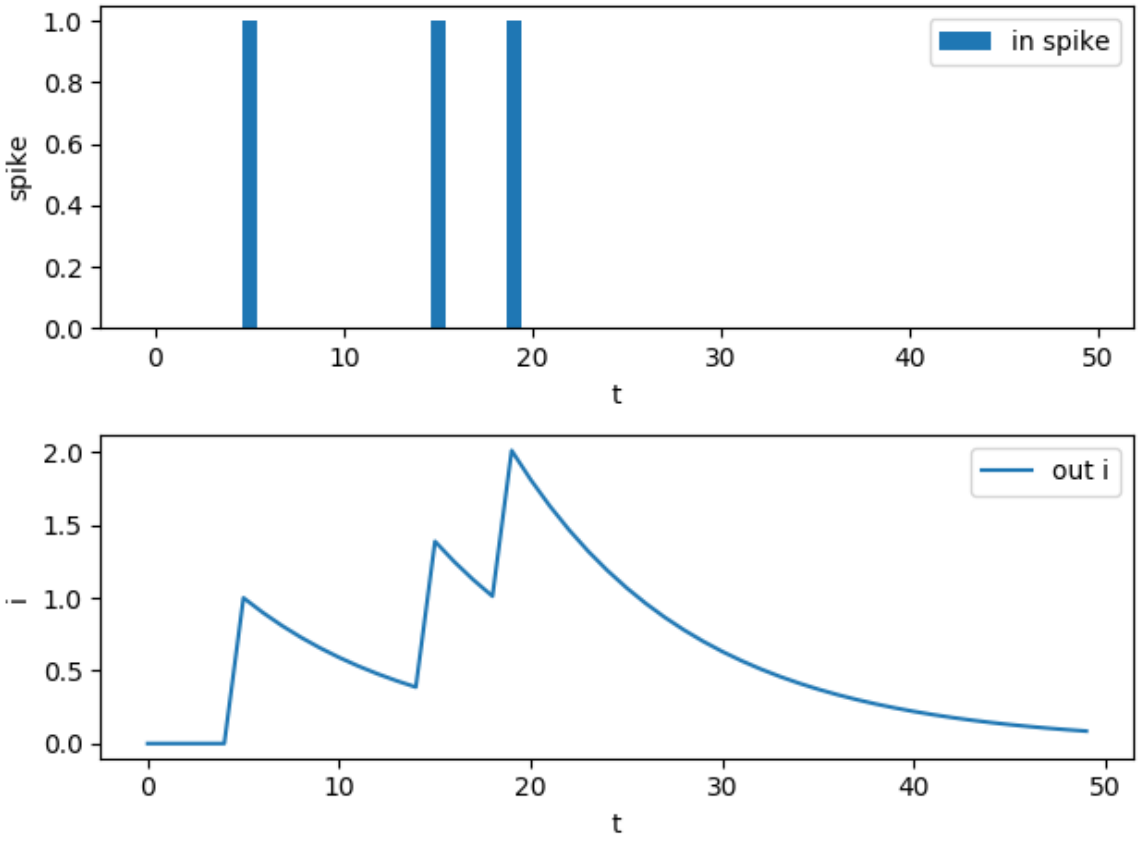
$$I(t) = I(t) + 1$$

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):
    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: Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**ChannelsPool**(*pool: MaxPool1d*)

基类：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: 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: Module = ReLU()*)

基类：MemoryModule

- *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。

备注： 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: Tensor*) → Tensor

extra_repr() → str

training: bool

class spikingjelly.clock_driven.layer.**MultiStepContainer**(**args*)

基类：Sequential

- *API in English*

参数
args (*torch.nn.Module*) –单个或多个网络模块

将单步模块包装成多步模块的包装器。

小技巧： 阅读传播模式 以获取更多关于单步和多步传播的信息。

- *中文 API*

参数
args (*torch.nn.Module*) –one or many modules

A container that wraps single-step modules to a multi-step modules.

Tip

Read *Propagation Pattern* for more details about single-step and multi-step propagation.

---

forward(*x_seq: Tensor*)

参数
x_seq (*Tensor*) –shape=[T, batch_size, ⋯]

返回
y_seq, shape=[T, batch_size, ⋯]

返回类型
Tensor

training: bool

class spikingjelly.clock_driven.layer.**SeqToANNContainer**(**args*)

基类：Sequential

- *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: Tensor*)

参数
x_seq (*Tensor*) –shape=[T, batch_size, ···]

返回
y_seq, shape=[T, batch_size, ···]

返回类型
Tensor

training: bool

class spikingjelly.clock_driven.layer.**STDPLearner**(*tau_pre: float*, *tau_post: float*, *f_pre*, *f_post*)

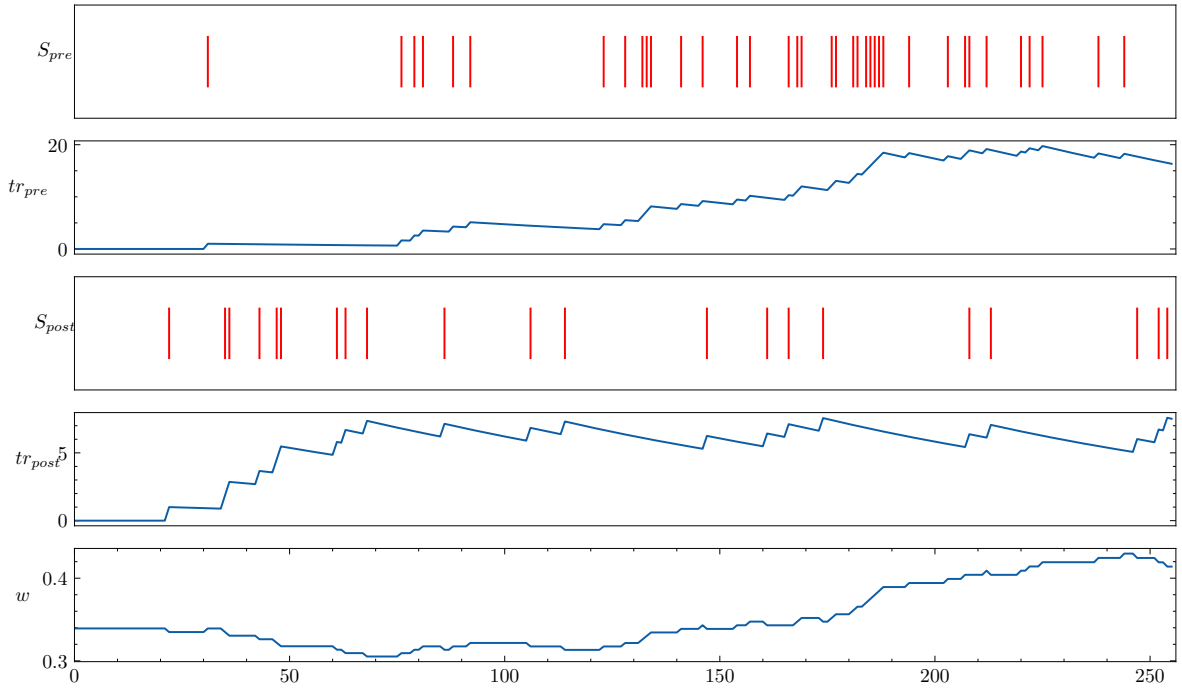
基类：MemoryModule

```
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 = []
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)
```

(续下页)

(接上页)

```
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)
plt.subplot(5, 1, 5)
plt.plot(t, w)
plt.ylabel('$w$', rotation=0, labelpad=10)
plt.xlim(0, T)

plt.show()
```



stdp(*s_pre: Tensor*, *s_post: Tensor*, *module: Module*, *learning_rate: float*)

training: bool

class spikingjelly.clock_driven.layer.**PrintShapeModule**(*ext_str='PrintShapeModule'*)

基类：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.

forward(*x: Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**ConvBatchNorm2d**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int, int]]*, *stride: Union[int, Tuple[int, int]] = 1*, *padding: Union[int, Tuple[int, int]] = 0*, *dilation: Union[int, Tuple[int, int]] = 1*, *groups: int = 1*, *padding_mode: str = 'zeros'*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*)

基类：Module

A fused Conv2d-BatchNorm2d module. See torch.nn.Conv2d and torch.nn.BatchNorm2d for params information.

Examples:

```
convbn = ConvBatchNorm2d(3, 64, kernel_size=3, padding=1)
x = torch.rand([16, 3, 224, 224])
with torch.no_grad():
    convbn.eval()
    conv = convbn.get_fused_conv()
    conv.eval()
    print((convbn(x) - conv(x)).abs().max())
```

(续下页)

(接上页)

```
    k_weight = 1.5
    b_weight = 0.4
    k_bias = 0.8
    b_bias = 0.1

    conv.weight.data *= k_weight
    conv.weight.data += b_weight
    conv.bias.data *= k_bias
    conv.bias.data += b_bias

    convbn.scale_fused_weight(k_weight, b_weight)
    convbn.scale_fused_bias(k_bias, b_bias)

    print((convbn(x) - conv(x)).abs().max())
```

forward(*x: Tensor*)

get_fused_weight()

返回

the weight of this fused module

返回类型

Tensor

get_fused_bias()

返回

the bias of this fused module

返回类型

Tensor

scale_fused_weight(*k=None*, *b=None*)

参数

- **k** (*float or None*) –scale factor
- **b** (*float or None*) –bias factor

Set the *weight* of this fused module to *weight * k + b*

scale_fused_bias(*k=None*, *b=None*)

参数

- **k** (*float or None*) –scale factor

- **b** (*float or None*) –bias factor

Set the *bias* of this fused module to *bias* * *k* + *b*

get_fused_conv()

training: bool

class spikingjelly.clock_driven.layer.**ElementWiseRecurrentContainer**(*sub_module: Module*, *element_wise_function: Callable*)

基类：MemoryModule

参数

- **sub_module** (*torch.nn.Module*) –the contained module
- **element_wise_function** (*Callable*) –the user-defined element-wise function, which should have the format `z=f(x, y)`

A container that use a element-wise recurrent connection. Denote the inputs and outputs of `sub_module` as $i[t]$ and $y[t]$ (Note that $y[t]$ is also the outputs of this module), and the inputs of this module as $x[t]$, then

$$i[t] = f(x[t], y[t-1])$$

where f is the user-defined element-wise function. We set $y[-1] = 0$.

Note

The shape inputs and outputs of `sub_module` must be the same.

Codes example:

```
T = 8
net = ElementWiseRecurrentContainer(neuron.IFNode(v_reset=None), element_wise_
↪function=lambda x, y: x + y)
print(net)
x = torch.zeros([T])
x[0] = 1.5
for t in range(T):
    print(t, f'x[t]={x[t]}, s[t]={net(x[t])}')

functional.reset_net(net)
```

forward(*x: Tensor*)

extra_repr() → str

training: bool

class spikingjelly.clock_driven.layer.**LinearRecurrentContainer**(*sub_module: Module*, *in_features: int*, *out_features: int*, *bias: bool = True*)

基类：MemoryModule

参数

- **sub_module** (*torch.nn.Module*) –the contained module
- **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

A container that use a linear recurrent connection. Denote the inputs and outputs of `sub_module` as $i[t]$ and $y[t]$ (Note that $y[t]$ is also the outputs of this module), and the inputs of this module as $x[t]$, then

$$i[t] = \begin{pmatrix} x[t] \\ y[t-1] \end{pmatrix} W^T + b$$

where W, b are the weight and bias of the linear connection. We set $y[-1] = 0$.

$x[t]$ should have the shape `[N, *, in_features]`, and $y[t]$ has the shape `[N, *, out_features]`.

Tip

The recurrent connection is implement by `torch.nn.Linear(in_features + out_features, in_features, bias)`.

```
in_features = 4
out_features = 2
T = 8
N = 2
net = LinearRecurrentContainer(
    nn.Sequential(
        nn.Linear(in_features, out_features),
        neuron.LIFNode(),
    ),
    in_features, out_features)
print(net)
x = torch.rand([T, N, in_features])
```

(续下页)

(接上页)

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

functional.reset_net(net)
```

forward(*x: Tensor*)

training: bool

class spikingjelly.clock_driven.layer.**MultiStepThresholdDependentBatchNorm1d**(*alpha: float*, *v_th: float*, **args*, ***kwargs*)

基类：_MultiStepThresholdDependentBatchNormBase

- *API in English*

参数

- **alpha** (*float*) –由网络结构决定的超参数
- **v_th** (*float*) –下一个脉冲神经元层的阈值

*args, **kwargs 中的参数与 torch.nn.BatchNorm1d 的参数相同。

Going Deeper With Directly-Trained Larger Spiking Neural Networks 一文提出的 Threshold-Dependent Batch Normalization (tdBN)。

- 中文 *API*

参数

- **alpha** (*float*) –the hyper-parameter depending on network structure
- **v_th** (*float*) –the threshold of next spiking neurons layer

Other parameters in *args, **kwargs are same with those of torch.nn.BatchNorm1d.

The Threshold-Dependent Batch Normalization (tdBN) proposed in Going Deeper With Directly-Trained Larger Spiking Neural Networks.

num_features: int

eps: float

momentum: float

affine: **bool**

track_running_stats: **bool**

class spikingjelly.clock_driven.layer.**MultiStepThresholdDependentBatchNorm2d**(*alpha: float*, *v_th: float*, **args*, ***kwargs*)

基类：_MultiStepThresholdDependentBatchNormBase

- *API in English*

参数

- **alpha** (*float*) –由网络结构决定的超参数
- **v_th** (*float*) –下一个脉冲神经元层的阈值

*args, **kwargs 中的参数与 torch.nn.BatchNorm2d 的参数相同。

Going Deeper With Directly-Trained Larger Spiking Neural Networks 一文提出的 Threshold-Dependent Batch Normalization (tdBN)。

- 中文 *API*

参数

- **alpha** (*float*) –the hyper-parameter depending on network structure
- **v_th** (*float*) –the threshold of next spiking neurons layer

Other parameters in *args, **kwargs are same with those of torch.nn.BatchNorm2d.

The Threshold-Dependent Batch Normalization (tdBN) proposed in Going Deeper With Directly-Trained Larger Spiking Neural Networks.

num_features: **int**

eps: **float**

momentum: **float**

affine: **bool**

track_running_stats: **bool**

class spikingjelly.clock_driven.layer.**MultiStepThresholdDependentBatchNorm3d**(*alpha: float*, *v_th: float*, **args*, ***kwargs*)

基类：_MultiStepThresholdDependentBatchNormBase

- *API in English*

参数

- **alpha** (*float*) –由网络结构决定的超参数
- **v_th** (*float*) –下一个脉冲神经元层的阈值

*args, **kwargs 中的参数与 torch.nn.BatchNorm3d 的参数相同。

Going Deeper With Directly-Trained Larger Spiking Neural Networks 一文提出的 Threshold-Dependent Batch Normalization (tdBN)。

- 中文 *API*

参数

- **alpha** (*float*) –the hyper-parameter depending on network structure
- **v_th** (*float*) –the threshold of next spiking neurons layer

Other parameters in *args, **kwargs are same with those of torch.nn.BatchNorm3d.

The Threshold-Dependent Batch Normalization (tdBN) proposed in Going Deeper With Directly-Trained Larger Spiking Neural Networks.

num_features: int

eps: float

momentum: float

affine: bool

track_running_stats: bool

class spikingjelly.clock_driven.layer.**MultiStepTemporalWiseAttention**(*T: int*, *reduction: int = 16*, *dimension: int = 4*)

基类：Module

- *API in English*

参数

- **T** –输入数据的时间步长
- **reduction** –压缩比
- **dimension** –输入数据的维度。当输入数据为 [T, N, C, H, W] 时，dimension = 4；输入数据维度为 [T, N, L] 时，dimension = 2。

Temporal-Wise Attention Spiking Neural Networks for Event Streams Classification 中提出的 MultiStepTemporalWiseAttention 层。MultiStepTemporalWiseAttention 层必须放在二维卷积层之后脉冲神经元之前，例如：

```
Conv2d -> MultiStepTemporalWiseAttention -> LIF
```

输入的尺寸是 `[T, N, C, H, W]` 或者 `[T, N, L]` ，经过 MultiStepTemporalWiseAttention 层，输出为 `[T, N, C, H, W]` 或者 `[T, N, L]` 。

`reduction` 是压缩比，相当于论文中的 r。

- *中文 API*

参数

- **T** –timewindows of input
- **reduction** –reduction ratio
- **dimension** –Dimensions of input. If the input dimension is [T, N, C, H, W], dimension = 4; when the input dimension is [T, N, L], dimension = 2.

The MultiStepTemporalWiseAttention layer is proposed in Temporal-Wise Attention Spiking Neural Networks for Event Streams Classification.

It should be placed after the convolution layer and before the spiking neurons, e.g.,

```
Conv2d -> MultiStepTemporalWiseAttention -> LIF
```

The dimension of the input is `[T, N, C, H, W]` or `[T, N, L]` , after the MultiStepTemporalWiseAttention layer, the output dimension is `[T, N, C, H, W]` or `[T, N, L]` .

`reduction` is the reduction ratio， which is r in the paper.

training: bool

forward(*x_seq: Tensor*)

spikingjelly.clock_driven.neuron package

Module contents

spikingjelly.clock_driven.neuron.**check_backend**(*backend: str*)

class spikingjelly.clock_driven.neuron.**BaseNode**(*v_threshold: float = 1.0*, *v_reset: float = 0.0*, *surrogate_function: Callable = Sigmoid(alpha=4.0, spiking=True)*, *detach_reset: bool = False*)

基类：MemoryModule

- *API in English*

参数

- **v_threshold** (*float*) –神经元的阈值电压
- **v_reset** (*float*) –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** (*Callable*) –反向传播时用来计算脉冲函数梯度的替代函数
- **detach_reset** (*bool*) –是否将 reset 过程的计算图分离

可微分 SNN 神经元的基类神经元。

- 中文 *API*

参数

- **v_threshold** (*float*) –threshold voltage of neurons
- **v_reset** (*float*) –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** (*Callable*) –surrogate function for replacing gradient of spiking functions during back-propagation
- **detach_reset** (*bool*) –whether detach the computation graph of reset

This class is the base class of differentiable spiking neurons.

abstract neuronal_charge(*x: 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(*spike*)

- *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()

forward(*x: Tensor*)

- *API in English*

参数

x (*torch.Tensor*) –输入到神经元的电压增量

返回

神经元的输出脉冲

返回类型

torch.Tensor

按照充电、放电、重置的顺序进行前向传播。

- 中文 *API*

参数

x (*torch.Tensor*) –increment of voltage inputted to neurons

返回

out spikes of neurons

返回类型

torch.Tensor

Forward by the order of *neuronal_charge*, *neuronal_fire*, and *neuronal_reset*.

training: bool

class spikingjelly.clock_driven.neuron.**AdaptiveBaseNode**(*v_threshold: float = 1.0*, *v_reset: float = 0.0*, *v_rest: float = 0.0*, *w_rest: float = 0*, *tau_w: float = 2.0*, *a: float = 0.0*, *b: float = 0.0*, *surrogate_function: Callable = Sigmoid(alpha=4.0, spiking=True)*, *detach_reset: bool = False*)

基类：*BaseNode*

neuronal_adaptation(*spike*)

extra_repr()

forward(*x: torch.Tensor*)

Helper for @overload to raise when called.

training: bool

class spikingjelly.clock_driven.neuron.**IFNode**(*v_threshold: float = 1.0*, *v_reset: float = 0.0*, *surrogate_function: Callable = Sigmoid(alpha=4.0, spiking=True)*, *detach_reset: bool = False*, *cupy_fp32_inference=False*)

基类：*BaseNode*

- *API in English*

参数

- **v_threshold** (*float*) –神经元的阈值电压
- **v_reset** (*float*) –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** (*Callable*) –反向传播时用来计算脉冲函数梯度的替代函数
- **detach_reset** (*bool*) –是否将 reset 过程的计算图分离
- **cupy_fp32_inference** (*bool*) –若为 *True*，在 *eval* 模式下，使用 float32，却在 GPU 上运行，并且 *cupy* 已经安装，则会自动使用 *cupy* 进行加速

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：

$$V[t] = V[t-1] + X[t]$$

- 中文 *API*

参数

- **v_threshold** (*float*) –threshold voltage of neurons
- **v_reset** (*float*) –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** (*Callable*) –surrogate function for replacing gradient of spiking functions during back-propagation
- **detach_reset** (*bool*) –whether detach the computation graph of reset
- **cupy_fp32_inference** (*bool*) –If *True*, if this module is in *eval* mode, using float32, running on GPU, and *cupy* is installed, then this module will use *cupy* to accelerate

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:

$$V[t] = V[t-1] + X[t]$$

neuronal_charge(*x: Tensor*)

forward(*x: Tensor*)

training: bool

class spikingjelly.clock_driven.neuron.**MultiStepIFNode**(*v_threshold: float = 1.0*, *v_reset: float = 0.0*, *surrogate_function: Callable = Sigmoid(alpha=4.0, spiking=True)*, *detach_reset: bool = False*, *backend='torch'*, *lava_s_cale=64*)

基类：*IFNode*

- *API in English*

参数

- **v_threshold** (*float*) –神经元的阈值电压
- **v_reset** (*float*) –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** (*Callable*) –反向传播时用来计算脉冲函数梯度的替代函数

- **detach_reset** (*bool*) –是否将 reset 过程的计算图分离
- **backend** (*str*) –使用哪种计算后端，可以为 `'torch'` 或 `'cupy'`。`'cupy'` 速度更快，但仅支持 GPU。

多步版本的*spikingjelly.clock_driven.neuron.IFNode*。

小技巧： 对于多步神经元，输入 `x_seq.shape = [T, *]`，不仅可以使用 `.v` 和 `.spike` 获取 `t = T - 1` 时刻的电压和脉冲，还能够使用 `.v_seq` 和 `.spike_seq` 获取完整的 `T` 个时刻的电压和脉冲。

小技巧： 阅读传播模式 以获取更多关于单步和多步传播的信息。

- 中文 *API*

参数

- **v_threshold** (*float*) –threshold voltage of neurons
- **v_reset** (*float*) –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** (*Callable*) –surrogate function for replacing gradient of spiking functions during back-propagation
- **detach_reset** (*bool*) –whether detach the computation graph of reset
- **backend** (*str*) –use which backend, `'torch'` or `'cupy'`. `'cupy'` is faster but only supports GPU

The multi-step version of *spikingjelly.clock_driven.neuron.IFNode*.

Tip

The input for multi-step neurons are `x_seq.shape = [T, *]`. We can get membrane potential and spike at time-step `t = T - 1` by `.v` and `.spike`. We can also get membrane potential and spike at all `T` time-steps by `.v_seq` and `.spike_seq`.

Tip

Read *Propagation Pattern* for more details about single-step and multi-step propagation.

---

forward(*x_seq: Tensor*)

extra_repr()

to_lava()

reset()

training: bool

class spikingjelly.clock_driven.neuron.**LIFNode**(*tau: float = 2.0*, *decay_input: bool = True*, *v_threshold: float = 1.0*, *v_reset: float = 0.0*, *surrogate_function: Callable = Sigmoid(alpha=4.0, spiking=True)*, *detach_reset: bool = False*, *cupy_fp32_inference=False*)

基类：*BaseNode*

- *API in English*

参数

- **tau** (*float*) –膜电位时间常数
- **decay_input** (*bool*) –输入是否会衰减
- **v_threshold** (*float*) –神经元的阈值电压
- **v_reset** (*float*) –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** (*Callable*) –反向传播时用来计算脉冲函数梯度的替代函数
- **detach_reset** (*bool*) –是否将 reset 过程的计算图分离
- **cupy_fp32_inference** (*bool*) –若为 *True*，在 *eval* 模式下，使用 float32，却在 GPU 上运行，并且 *cupy* 已经安装，则会自动使用 *cupy* 进行加速

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：

若 decay_input == True:

$$V[t] = V[t-1] + \frac{1}{\tau}(X[t] - (V[t-1] - V_{reset}))$$

若 decay_input == False:

$$V[t] = V[t-1] - \frac{1}{\tau}(V[t-1] - V_{reset}) + X[t]$$

小技巧： 在 *eval* 模式下，使用 float32，却在 GPU 上运行，并且 *cupy* 已经安装，则会自动使用 *cupy* 进行加速。

- 中文 *API*

参数

- **tau** (*float*) –membrane time constant
- **decay_input** (*bool*) –whether the input will decay
- **v_threshold** (*float*) –threshold voltage of neurons
- **v_reset** (*float*) –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** (*Callable*) –surrogate function for replacing gradient of spiking functions during back-propagation
- **detach_reset** (*bool*) –whether detach the computation graph of reset
- **cupy_fp32_inference** (*bool*) –If *True*, if this module is in *eval* mode, using float32, running on GPU, and *cupy* is installed, then this module will use *cupy* to accelerate

The Leaky Integrate-and-Fire neuron, which can be seen as a leaky integrator. The subthreshold neural dynamics of it is as followed:

IF `decay_input == True`:

$$V[t] = V[t-1] + \frac{1}{\tau}(X[t] - (V[t-1] - V_{reset}))$$

IF `decay_input == False`:

$$V[t] = V[t-1] - \frac{1}{\tau}(V[t-1] - V_{reset}) + X[t]$$

Tip

If this module is in *eval* mode, using float32, running on GPU, and *cupy* is installed, then this module will use *cupy* to accelerate.

extra_repr()

neuronal_charge(*x: Tensor*)

forward(*x: Tensor*)

training: bool

class spikingjelly.clock_driven.neuron.**MultiStepLIFNode**(*tau: float = 2.0, decay_input: bool = True, v_threshold: float = 1.0, v_reset: float = 0.0, surrogate_function: Callable = Sigmoid(alpha=4.0, spiking=True), detach_reset: bool = False, backend='torch', lava_s_cale=64*)

基类：*LIFNode*

- *API in English*

参数

- **tau** (*float*) –膜电位时间常数
- **decay_input** (*bool*) –输入是否会衰减
- **v_threshold** (*float*) –神经元的阈值电压
- **v_reset** (*float*) –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** (*Callable*) –反向传播时用来计算脉冲函数梯度的替代函数
- **detach_reset** (*bool*) –是否将 reset 过程的计算图分离
- **backend** (*str*) –使用哪种计算后端，可以为 'torch' 或 'cupy'。'cupy' 速度更快，但仅支持 GPU。

多步版本的*spikingjelly.clock_driven.neuron.LIFNode*。

小技巧： 对于多步神经元，输入 x_seq.shape = [T, *]，不仅可以使用 .v 和 .spike 获取 t = T - 1 时刻的电压和脉冲，还能够使用 .v_seq 和 .spike_seq 获取完整的 T 个时刻的电压和脉冲。

小技巧： 阅读传播模式 以获取更多关于单步和多步传播的信息。

- 中文 *API*

参数

- **tau** (*float*) –membrane time constant
- **decay_input** (*bool*) –whether the input will decay
- **v_threshold** (*float*) –threshold voltage of neurons
- **v_reset** (*float*) –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** (*Callable*) –surrogate function for replacing gradient of spiking functions during back-propagation
- **detach_reset** (*bool*) –whether detach the computation graph of reset
- **backend** (*str*) –use which backend, `'torch'` or `'cupy'`. `'cupy'` is faster but only supports GPU

The multi-step version of *spikingjelly.clock_driven.neuron.LIFNode*.

Tip

The input for multi-step neurons are `x_seq.shape = [T, *]`. We can get membrane potential and spike at time-step `t = T - 1` by `.v` and `.spike`. We can also get membrane potential and spike at all `T` time-steps by `.v_seq` and `.spike_seq`.

Tip

Read *Propagation Pattern* for more details about single-step and multi-step propagation.

---

forward(*x_seq: Tensor*)

extra_repr()

to_lava()

reset()

training: bool

class spikingjelly.clock_driven.neuron.**ParametricLIFNode**(*init_tau: float = 2.0*, *decay_input: bool = True*, *v_threshold: float = 1.0*, *v_reset: float = 0.0*, *surrogate_function: Callable = Sigmoid(alpha=4.0, spiking=True)*, *detach_reset: bool = False*)

基类：*BaseNode*

- *API in English*

参数

- **init_tau** (*float*) –膜电位时间常数的初始值
- **decay_input** (*bool*) –输入是否会衰减
- **v_threshold** (*float*) –神经元的阈值电压
- **v_reset** (*float*) –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** (*Callable*) –反向传播时用来计算脉冲函数梯度的替代函数
- **detach_reset** (*bool*) –是否将 reset 过程的计算图分离

Incorporating Learnable Membrane Time Constant to Enhance Learning of Spiking Neural Networks 提出的 Parametric Leaky Integrate-and-Fire (PLIF) 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：

若 decay_input == True:

$$V[t] = V[t-1] + \frac{1}{\tau}(X[t] - (V[t-1] - V_{reset}))$$

若 decay_input == False:

$$V[t] = V[t-1] - \frac{1}{\tau}(V[t-1] - V_{reset}) + X[t]$$

其中 $\frac{1}{\tau} = \text{Sigmoid}(w)$，$w$ 是可学习的参数。

- 中文 *API*

参数

- **init_tau** (*float*) –the initial value of membrane time constant

- **decay_input** (*bool*) –whether the input will decay
- **v_threshold** (*float*) –threshold voltage of neurons
- **v_reset** (*float*) –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** (*Callable*) –surrogate function for replacing gradient of spiking functions during back-propagation
- **detach_reset** (*bool*) –whether detach the computation graph of reset

The Parametric Leaky Integrate-and-Fire (PLIF) neuron, which is proposed by Incorporating Learnable Membrane Time Constant to Enhance Learning of Spiking Neural Networks and can be seen as a leaky integrator. The subthreshold neural dynamics of it is as followed:

IF `decay_input == True`:

$$V[t] = V[t-1] + \frac{1}{\tau}(X[t] - (V[t-1] - V_{reset}))$$

IF `decay_input == False`:

$$V[t] = V[t-1] - \frac{1}{\tau}(V[t-1] - V_{reset}) + X[t]$$

where $\frac{1}{\tau} = \text{Sigmoid}(w)$, w is a learnable parameter.

extra_repr()

neuronal_charge(*x: Tensor*)

training: bool

class spikingjelly.clock_driven.neuron.**MultiStepParametricLIFNode**(*init_tau: float = 2.0, decay_input: bool = True, v_threshold: float = 1.0, v_reset: float = 0.0, surrogate_function: Callable = Sigmoid(alpha=4.0, spiking=True), detach_reset: bool = False, backend='torch'*)

基类：*ParametricLIFNode*

- *API in English*

参数

- **init_tau** (*float*) –膜电位时间常数的初始值
- **decay_input** (*bool*) –输入是否会衰减
- **v_threshold** (*float*) –神经元的阈值电压
- **v_reset** (*float*) –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** (*Callable*) –反向传播时用来计算脉冲函数梯度的替代函数
- **detach_reset** (*bool*) –是否将 reset 过程的计算图分离

多步版本的 Incorporating Learnable Membrane Time Constant to Enhance Learning of Spiking Neural Networks 提出的 Parametric Leaky Integrate-and-Fire (PLIF) 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：

$$V[t] = V[t-1] + \frac{1}{\tau}(X[t] - (V[t-1] - V_{reset}))$$

其中 $\frac{1}{\tau} = \text{Sigmoid}(w)$，$w$ 是可学习的参数。

小技巧： 对于多步神经元，输入 x_seq.shape = [T, *]，不仅可以使用 .v 和 .spike 获取 t = T - 1 时刻的电压和脉冲，还能够使用 .v_seq 和 .spike_seq 获取完整的 T 个时刻的电压和脉冲。

小技巧： 阅读传播模式 以获取更多关于单步和多步传播的信息。

- 中文 *API*

参数

- **init_tau** (*float*) –the initial value of membrane time constant
- **decay_input** (*bool*) –whether the input will decay
- **v_threshold** (*float*) –threshold voltage of neurons
- **v_reset** (*float*) –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** (*Callable*) –surrogate function for replacing gradient of spiking functions during back-propagation
- **detach_reset** (*bool*) –whether detach the computation graph of reset
- **backend** (*str*) –use which backend, `'torch'` or `'cupy'`. `'cupy'` is faster but only supports GPU

The multi-step Parametric Leaky Integrate-and-Fire (PLIF) neuron, which is proposed by Incorporating Learnable Membrane Time Constant to Enhance Learning of Spiking Neural Networks and can be seen as a leaky integrator. The subthreshold neural dynamics of it is as followed:

$$V[t] = V[t-1] + \frac{1}{\tau}(X[t] - (V[t-1] - V_{reset})$$

where $\frac{1}{\tau} = \text{Sigmoid}(w)$, w is a learnable parameter.

Tip

The input for multi-step neurons are `x_seq.shape = [T, *]`. We can get membrane potential and spike at time-step `t = T - 1` by `.v` and `.spike`. We can also get membrane potential and spike at all `T` time-steps by `.v_seq` and `.spike_seq`.

Tip

Read *Propagation Pattern* for more details about single-step and multi-step propagation.

---

forward(*x_seq: Tensor*)

extra_repr()

training: bool

class spikingjelly.clock_driven.neuron.**QIFNode**(*tau: float = 2.0, v_c: float = 0.8, a0: float = 1.0, v_threshold: float = 1.0, v_rest: float = 0.0, v_reset: float = -0.1, surrogate_function: Callable = Sigmoid(alpha=4.0, spiking=True), detach_reset: bool = False*)

基类：*BaseNode*

- *API in English*

参数

- **tau** (*float*) –膜电位时间常数
- **v_c** (*float*) –关键电压

- **a0** (*float*) –
- **v_threshold** (*float*) –神经元的阈值电压
- **v_rest** (*float*) –静息电位
- **v_reset** (*float*) –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** (*Callable*) –反向传播时用来计算脉冲函数梯度的替代函数
- **detach_reset** (*bool*) –是否将 reset 过程的计算图分离

Quadratic Integrate-and-Fire 神经元模型，一种非线性积分发放神经元模型，也是指数积分发放神经元 (Exponential Integrate-and-Fire) 的近似版本。其阈下神经动力学方程为：

$$V[t] = V[t-1] + \frac{1}{\tau}(X[t] + a_0(V[t-1] - V_{rest})(V[t-1] - V_c))$$

- 中文 *API*

参数

- **tau** (*float*) –membrane time constant
- **v_c** (*float*) –critical voltage
- **a0** (*float*) –
- **v_threshold** (*float*) –threshold voltage of neurons
- **v_rest** (*float*) –resting potential
- **v_reset** (*float*) –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** (*Callable*) –surrogate function for replacing gradient of spiking functions during back-propagation
- **detach_reset** (*bool*) –whether detach the computation graph of reset

The Quadratic Integrate-and-Fire neuron is a kind of nonlinear integrate-and-fire models and also an approximation of the Exponential Integrate-and-Fire model. The subthreshold neural dynamics of it is as followed:

$$V[t] = V[t-1] + \frac{1}{\tau}(X[t] + a_0(V[t-1] - V_{rest})(V[t-1] - V_c))$$

extra_repr()

neuronal_charge(*x: Tensor*)

training: bool

class spikingjelly.clock_driven.neuron.**EIFNode**(*tau: float = 2.0*, *delta_T: float = 1.0*, *theta_rh: float = 0.8*, *v_threshold: float = 1.0*, *v_rest: float = 0.0*, *v_reset: float = -0.1*, *surrogate_function: Callable = Sigmoid(alpha=4.0, spiking=True)*, *detach_reset: bool = False*)

基类：*BaseNode*

- *API in English*

参数

- **tau** (*float*) –膜电位时间常数
- **delta_T** (*float*) –陡峭度参数
- **theta_rh** (*float*) –基强度电压阈值
- **v_threshold** (*float*) –神经元的阈值电压
- **v_rest** (*float*) –静息电位
- **v_reset** (*float*) –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** (*Callable*) –反向传播时用来计算脉冲函数梯度的替代函数
- **detach_reset** (*bool*) –是否将 reset 过程的计算图分离

Exponential Integrate-and-Fire 神经元模型，一种非线性积分发放神经元模型，是由 HH 神经元模型 (Hodgkin-Huxley model) 简化后推导出的一维模型。在 $\Delta_T \to 0$ 时退化为 LIF 模型。其阈下神经动力学方程为：

$$V[t] = V[t-1] + \frac{1}{\tau}\left(X[t] - (V[t-1] - V_{rest}) + \Delta_T \exp\left(\frac{V[t-1] - \theta_{rh}}{\Delta_T}\right)\right)$$

- *中文 API*

参数

- **tau** (*float*) –membrane time constant
- **delta_T** (*float*) –sharpness parameter
- **theta_rh** (*float*) –rheobase threshold
- **v_threshold** (*float*) –threshold voltage of neurons
- **v_rest** (*float*) –resting potential

- **v_reset** (*float*) –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** (*Callable*) –surrogate function for replacing gradient of spiking functions during back-propagation
- **detach_reset** (*bool*) –whether detach the computation graph of reset

The Exponential Integrate-and-Fire neuron is a kind of nonlinear integrate-and-fire models and also an one-dimensional model derived from the Hodgkin-Huxley model. It degenerates to the LIF model when $\Delta_T \to 0$. The subthreshold neural dynamics of it is as followed:

$$V[t] = V[t-1] + \frac{1}{\tau}\left(X[t] - (V[t-1] - V_{rest}) + \Delta_T \exp\left(\frac{V[t-1] - \theta_{rh}}{\Delta_T}\right)\right)$$

extra_repr()

neuronal_charge(*x: Tensor*)

training: bool

class spikingjelly.clock_driven.neuron.**MultiStepEIFNode**(*tau: float = 2.0*, *delta_T: float = 1.0*, *theta_rh: float = 0.8*, *v_threshold: float = 1.0*, *v_rest: float = 0.0*, *v_reset: float = -0.1*, *surrogate_function: Callable = Sigmoid(alpha=4.0, spiking=True)*, *detach_reset: bool = False*, *backend='torch'*)

基类：*EIFNode*

- *API in English*

::param tau: 膜电位时间常数:type tau: float

参数

- **delta_T** (*float*) –陡峭度参数
- **theta_rh** (*float*) –基强度电压阈值
- **v_threshold** (*float*) –神经元的阈值电压
- **v_rest** (*float*) –静息电位
- **v_reset** (*float*) –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** (*Callable*) –反向传播时用来计算脉冲函数梯度的替代函数
- **detach_reset** (*bool*) –是否将 reset 过程的计算图分离

多步版本的*spikingjelly.clock_driven.neuron.EIFNode*。

对于多步神经元，输入 `x_seq.shape = [T, *]`，不仅可以使用 `.v` 和 `.spike` 获取 `t = T - 1` 时刻的电压和脉冲，还能够使用 `.v_seq` 和 `.spike_seq` 获取完整的 `T` 个时刻的电压和脉冲。

小技巧： 阅读传播模式 以获取更多关于单步和多步传播的信息。

- 中文 *API*

参数

- **tau** (*float*) –membrane time constant
- **delta_T** (*float*) –sharpness parameter
- **theta_rh** (*float*) –rheobase threshold
- **v_threshold** (*float*) –threshold voltage of neurons
- **v_rest** (*float*) –resting potential
- **v_reset** (*float*) –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** (*Callable*) –surrogate function for replacing gradient of spiking functions during back-propagation
- **detach_reset** (*bool*) –whether detach the computation graph of reset
- **backend** (*str*) –use which backend, `'torch'` or `'cupy'`. `'cupy'` is faster but only supports GPU

Tip

The input for multi-step neurons are `x_seq.shape = [T, *]`. We can get membrane potential and spike at time-step `t = T - 1` by `.v` and `.spike`. We can also get membrane potential and spike at all `T` time-steps by `.v_seq` and `.spike_seq`.

Tip

Read *Propagation Pattern* for more details about single-step and multi-step propagation.

---

forward(*x_seq: Tensor*)

extra_repr()

training: bool

class spikingjelly.clock_driven.neuron.**GeneralNode**(*a: float*, *b: float*, *c: float = 0.0*, *v_threshold: float = 1.0*, *v_reset: float = 0.0*, *surrogate_function: Callable = Sigmoid(alpha=4.0, spiking=True)*, *detach_reset: bool = False*)

基类：*BaseNode*

neuronal_charge(*x: Tensor*)

training: bool

class spikingjelly.clock_driven.neuron.**MultiStepGeneralNode**(*a: float*, *b: float*, *c: float*, *v_threshold: float = 1.0*, *v_reset: float = 0.0*, *surrogate_function: Callable = Sigmoid(alpha=4.0, spiking=True)*, *detach_reset: bool = False*, *backend='torch'*)

基类：*GeneralNode*

forward(*x_seq: Tensor*)

extra_repr()

training: bool

class spikingjelly.clock_driven.neuron.**LIAFNode**(*act: Callable*, *threshold_related: bool*, **args*, ***kwargs*)

基类：*LIFNode*

参数

- **act** (*Callable*) –the activation function
- **threshold_related** (*bool*) –whether the neuron uses threshold related (TR mode). If true, *y = act(h - v_th)*, otherwise *y = act(h)*

Other parameters in **args, **kwargs* are same with `LIFNode`.

The LIAF neuron proposed in LIAF-Net: Leaky Integrate and Analog Fire Network for Lightweight and Efficient Spatiotemporal Information Processing.

Warning

The outputs of this neuron are not binary spikes.

---

training: bool

forward(*x: Tensor*)

spikingjelly.clock_driven.model package

Submodules

spikingjelly.clock_driven.model.spiking_resnet module

class spikingjelly.clock_driven.model.spiking_resnet.**SpikingResNet**(*block*, *layers*, *num_classes=1000*, *zero_init_residual=False*, *groups=1*, *width_per_group=64*, *replace_stride_with_dilation=None*, *norm_layer=None*, *single_step_neuron: Optional[callable] = None*, ***kwargs*)

基类：Module

forward(*x*)

training: bool

spikingjelly.clock_driven.model.spiking_resnet.**spiking_resnet18**(*pretrained=False*, *progress=True*, *single_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **single_step_neuron** (*callable*) –a single-step neuron
- **kwargs** (*dict*) –kwargs for *single_step_neuron*

返回

Spiking ResNet-18

返回类型

torch.nn.Module

A spiking version of ResNet-18 model from "Deep Residual Learning for Image Recognition"

spikingjelly.clock_driven.model.spiking_resnet.**spiking_resnet34**(*pretrained=False*, *progress=True*, *single_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **single_step_neuron** (*callable*) –a single-step neuron
- **kwargs** (*dict*) –kwargs for *single_step_neuron*

返回

Spiking ResNet-34

返回类型

torch.nn.Module

A spiking version of ResNet-34 model from "Deep Residual Learning for Image Recognition"

spikingjelly.clock_driven.model.spiking_resnet.**spiking_resnet50**(*pretrained=False*, *progress=True*, *single_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **single_step_neuron** (*callable*) –a single-step neuron
- **kwargs** (*dict*) –kwargs for *single_step_neuron*

返回

Spiking ResNet-50

返回类型

torch.nn.Module

A spiking version of ResNet-50 model from "Deep Residual Learning for Image Recognition"

spikingjelly.clock_driven.model.spiking_resnet.**spiking_resnet101**(*pretrained=False*, *progress=True*, *single_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **single_step_neuron** (*callable*) –a single-step neuron
- **kwargs** (*dict*) –kwargs for *single_step_neuron*

返回

Spiking ResNet-101

返回类型

torch.nn.Module

A spiking version of ResNet-101 model from "Deep Residual Learning for Image Recognition"

spikingjelly.clock_driven.model.spiking_resnet.**spiking_resnet152**(*pretrained=False*, *progress=True*, *single_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet

- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **single_step_neuron** (*callable*) –a single step neuron
- **kwargs** (*dict*) –kwargs for *single_step_neuron*

返回

Spiking ResNet-152

返回类型

torch.nn.Module

A spiking version of ResNet-152 model from "Deep Residual Learning for Image Recognition"

spikingjelly.clock_driven.model.spiking_resnet.**spiking_resnext50_32x4d**(*pretrained=False*, *progress=True*, *single_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **single_step_neuron** (*callable*) –a single step neuron
- **kwargs** (*dict*) –kwargs for *single_step_neuron*

返回

Spiking ResNeXt-50 32x4d

返回类型

torch.nn.Module

A spiking version of ResNeXt-50 32x4d model from "Aggregated Residual Transformation for Deep Neural Networks"

spikingjelly.clock_driven.model.spiking_resnet.**spiking_resnext101_32x8d**(*pretrained=False*, *progress=True*, *single_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **single_step_neuron** (*callable*) –a single step neuron
- **kwargs** (*dict*) –kwargs for *single_step_neuron*

返回

Spiking ResNeXt-101 32x8d

返回类型

torch.nn.Module

A spiking version of ResNeXt-101 32x8d model from "Aggregated Residual Transformation for Deep Neural Networks"

spikingjelly.clock_driven.model.spiking_resnet.**spiking_wide_resnet50_2**(*pretrained=False*, *progress=True*, *single_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **single_step_neuron** (*callable*) –a single step neuron
- **kwargs** (*dict*) –kwargs for *single_step_neuron*

返回

Spiking Wide ResNet-50-2

返回类型

torch.nn.Module

A spiking version of Wide ResNet-50-2 model from "Wide Residual Networks"

The model is the same as ResNet except for the bottleneck number of channels which is twice larger in every block. The number of channels in outer 1x1 convolutions is the same, e.g. last block in ResNet-50 has 2048-512-2048

channels, and in Wide ResNet-50-2 has 2048-1024-2048.

spikingjelly.clock_driven.model.spiking_resnet.**spiking_wide_resnet101_2**(*pretrained=False*, *progress=True*, *single_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **single_step_neuron** (*callable*) –a single step neuron
- **kwargs** (*dict*) –kwargs for *single_step_neuron*

返回

Spiking Wide ResNet-101-2

返回类型

torch.nn.Module

A spiking version of Wide ResNet-101-2 model from "Wide Residual Networks"

The model is the same as ResNet except for the bottleneck number of channels which is twice larger in every block. The number of channels in outer 1x1 convolutions is the same, e.g. last block in ResNet-50 has 2048-512-2048 channels, and in Wide ResNet-50-2 has 2048-1024-2048.

class spikingjelly.clock_driven.model.spiking_resnet.**MultiStepSpikingResNet**(*block*, *layers*, *num_classes=1000*, *zero_init_residual=Fa*
groups=1, *width_per_group=64*, *replace_stride_with_dila*
norm_layer=None, *T: Optional[int] = None*, *multi_step_neuron: Optional[callable] = None*, ***kwargs*)

基类：Module

forward(*x*)

参数

x (*torch.Tensor*) –the input with *shape=[N, C, H, W]* or *[*, N, C, H, W]*

返回

output

返回类型

torch.Tensor

training: bool

spikingjelly.clock_driven.model.spiking_resnet.**multi_step_spiking_resnet18**(*pretrained=False*, *progress=True*, *T: Optional[int] = None*, *multi_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **T** (*int*) –total time-steps
- **multi_step_neuron** (*callable*) –a multi-step neuron
- **kwargs** (*dict*) –kwargs for *multi_step_neuron*

返回

Spiking ResNet-18

返回类型

torch.nn.Module

A multi-step spiking version of ResNet-18 model from "Deep Residual Learning for Image Recognition"

spikingjelly.clock_driven.model.spiking_resnet.**multi_step_spiking_resnet34**(*pretrained=False*, *progress=True*, *T: Optional[int] = None*, *multi_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **T** (*int*) –total time-steps
- **multi_step_neuron** (*callable*) –a multi-step neuron
- **kwargs** (*dict*) –kwargs for *multi_step_neuron*

返回

Spiking ResNet-34

返回类型

torch.nn.Module

A multi-step spiking version of ResNet-34 model from "Deep Residual Learning for Image Recognition"

spikingjelly.clock_driven.model.spiking_resnet.**multi_step_spiking_resnet50**(*pretrained=False*, *progress=True*, *T: Optional[int] = None*, *multi_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **T** (*int*) –total time-steps
- **multi_step_neuron** (*callable*) –a multi-step neuron
- **kwargs** (*dict*) –kwargs for *multi_step_neuron*

返回

Spiking ResNet-50

返回类型

torch.nn.Module

A multi-step spiking version of ResNet-50 model from "Deep Residual Learning for Image Recognition"

spikingjelly.clock_driven.model.spiking_resnet.**multi_step_spiking_resnet101**(*pretrained=False*, *progress=True*, *T: Optional[int] = None*, *multi_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **T** (*int*) –total time-steps
- **multi_step_neuron** (*callable*) –a multi-step neuron
- **kwargs** (*dict*) –kwargs for *multi_step_neuron*

返回

Spiking ResNet-101

返回类型

torch.nn.Module

A multi-step spiking version of ResNet-101 model from "Deep Residual Learning for Image Recognition"

spikingjelly.clock_driven.model.spiking_resnet.**multi_step_spiking_resnet152**(*pretrained=False*, *progress=True*, *T: Optional[int] = None*, *multi_step_neuron: Optional[callable] = None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **T** (*int*) –total time-steps
- **multi_step_neuron** (*callable*) –a multi-step neuron
- **kwargs** (*dict*) –kwargs for *multi_step_neuron*

返回

Spiking ResNet-152

返回类型

torch.nn.Module

A multi-step spiking version of ResNet-152 model from "Deep Residual Learning for Image Recognition"

spikingjelly.clock_driven.model.spiking_resnet.**multi_step_spiking_resnext50_32x4d**(*pretrained=F*, *progress=Tru*, *T: Optional[int] = None*, *multi_step_n*, *Optional[callab* = *None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **T** (*int*) –total time-steps
- **multi_step_neuron** (*callable*) –a multi-step neuron
- **kwargs** (*dict*) –kwargs for *multi_step_neuron*

返回

Spiking ResNeXt-50 32x4d

返回类型

torch.nn.Module

A multi-step spiking version of ResNeXt-50 32x4d model from "Aggregated Residual Transformation for Deep Neural Networks"

spikingjelly.clock_driven.model.spiking_resnet.**multi_step_spiking_resnext101_32x8d**(*pretrained=*, *progress=T*, *T: Optional[int] =*, *multi_step_ Optional[calla =*, *None*, ***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **T** (*int*) –total time-steps
- **multi_step_neuron** (*callable*) –a multi-step neuron
- **kwargs** (*dict*) –kwargs for *multi_step_neuron*

返回

Spiking ResNeXt-101 32x8d

返回类型

torch.nn.Module

A multi-step spiking version of ResNeXt-101 32x8d model from "Aggregated Residual Transformation for Deep Neural Networks"

spikingjelly.clock_driven.model.spiking_resnet.**multi_step_spiking_wide_resnet50_2**(*pretrained=F*
progress=Tru
T:
Optional[int]
=
None,
multi_step_ne
Optional[callab
=
None,
***kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **T** (*int*) –total time-steps
- **multi_step_neuron** (*callable*) –a multi-step neuron
- **kwargs** (*dict*) –kwargs for *multi_step_neuron*

返回

Spiking Wide ResNet-50-2

返回类型

torch.nn.Module

A multi-step spiking version of Wide ResNet-50-2 model from "Wide Residual Networks"

The model is the same as ResNet except for the bottleneck number of channels which is twice larger in every block. The number of channels in outer 1x1 convolutions is the same, e.g. last block in ResNet-50 has 2048-512-2048 channels, and in Wide ResNet-50-2 has 2048-1024-2048.

spikingjelly.clock_driven.model.spiking_resnet.**multi_step_spiking_wide_resnet101_2**(*pretrained=F progress=T T: Optional[int] = None, multi_step_ Optional[calla = None, **kwargs*)

参数

- **pretrained** (*bool*) –If True, the SNN will load parameters from the ANN pre-trained on ImageNet
- **progress** (*bool*) –If True, displays a progress bar of the download to stderr
- **T** (*int*) –total time-steps
- **multi_step_neuron** (*callable*) –a multi-step neuron
- **kwargs** (*dict*) –kwargs for *multi_step_neuron*

返回

Spiking Wide ResNet-101-2

返回类型

torch.nn.Module

A multi-step spiking version of Wide ResNet-101-2 model from "Wide Residual Networks"

The model is the same as ResNet except for the bottleneck number of channels which is twice larger in every block. The number of channels in outer 1x1 convolutions is the same, e.g. last block in ResNet-50 has 2048-512-2048 channels, and in Wide ResNet-50-2 has 2048-1024-2048.

Module contents

spikingjelly.clock_driven.monitor package

Module contents

class spikingjelly.clock_driven.monitor.**Monitor**(*net: 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' 字符串或者 torch.device 类型，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 string 'cpu', 'cuda', 'cuda:0' or torch.device, 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*) → 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*) → 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.rnn package

Module contents

spikingjelly.clock_driven.rnn.**bidirectional_rnn_cell_forward**(*cell: Module*, *cell_reverse: Module*, *x: Tensor*, *states: Tensor*, *states_reverse: 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：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

---

备注： 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: 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: 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：*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

---

备注: 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):
```

(续下页)

(接上页)

```
    h, c = rnn(input[t], (h, c))
    output.append(h)
print(output)
```

forward(*x: 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：*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: *SpikingRNNCellBase*

forward(*x: 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：*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：*SpikingRNNCellBase*

forward(*x: Tensor*, *h=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：*SpikingRNNBase*

static base_cell()

static states_num()

training: bool

spikingjelly.clock_driven.surrogate package

Module contents

spikingjelly.clock_driven.surrogate.**heaviside**(*x: 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*, **args*, ***kwargs*)

参数

- **primitive_function** (*callable*) –梯度替代函数的原函数
- **spiking_function** (*callable*) –梯度替代函数

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：

```
def s2nn_apply(x, alpha, beta):
    return surrogate.s2nn.apply(x, alpha, beta)

surrogate.check_manual_grad(surrogate.S2NN.primitive_function, s2nn_apply,␣
↪alpha=4., beta=1.)
```

spikingjelly.clock_driven.surrogate.**check_cuda_grad**(*neu: Module*, *surrogate_function*, *device*, **args*, ***kwargs*)

class spikingjelly.clock_driven.surrogate.**SurrogateFunctionBase**(*alpha*, *spiking=True*)

基类：Module

set_spiking_mode(*spiking: bool*)

extra_repr()

static spiking_function(*x*, *alpha*)

static primitive_function(*x*, *alpha*)

cuda_code(*x: str*, *y: str*, *dtype='fp32'*)

cuda_code_start_comments()

cuda_code_end_comments()

forward(*x: Tensor*)

training: bool

class spikingjelly.clock_driven.surrogate.**MultiArgsSurrogateFunctionBase**(*spiking: bool*, **args*, ***kwargs*)

基类：Module

set_spiking_mode(*spiking: bool*)

cuda_code(*x: str*, *y: str*, *dtype='fp32'*)

cuda_code_start_comments()

cuda_code_end_comments()

training: bool

class spikingjelly.clock_driven.surrogate.**piecewise_quadratic**

基类：Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**PiecewiseQuadratic**(*alpha=1.0*, *spiking=True*)

基类：*SurrogateFunctionBase*

- *API in English*

参数

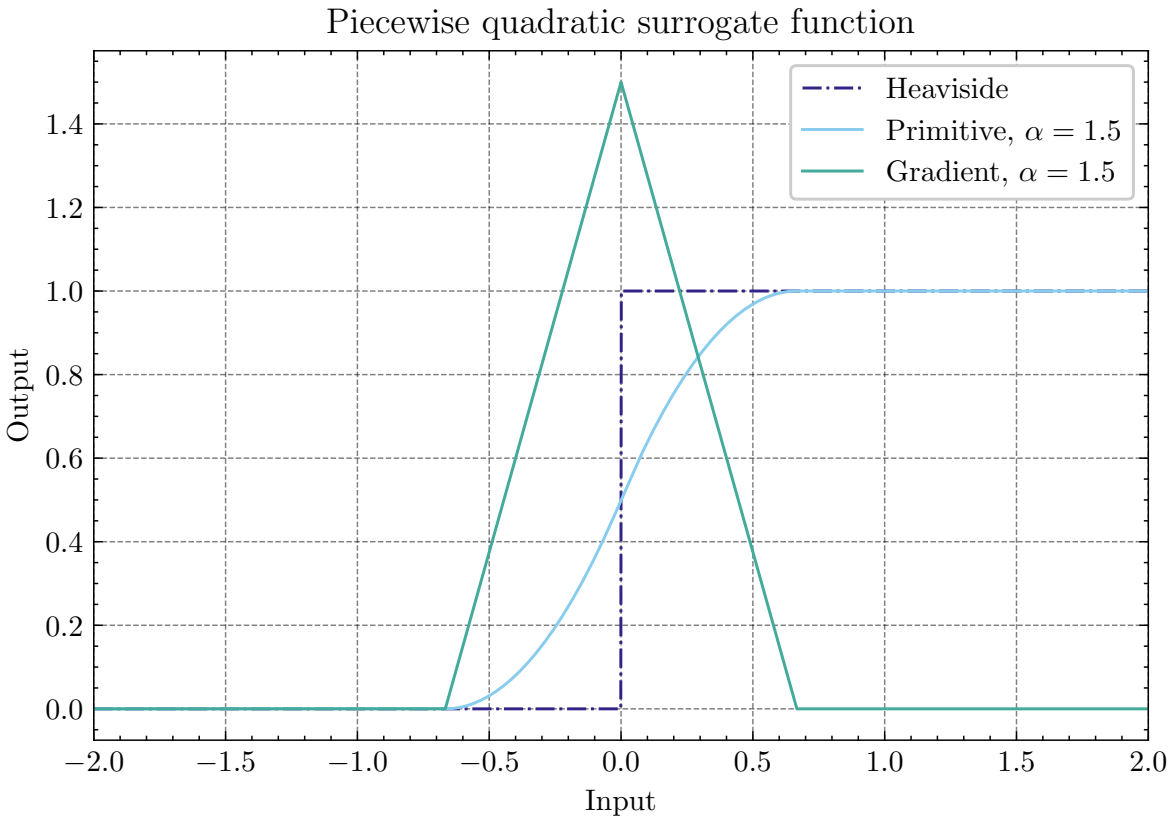
- **alpha** –控制反向传播时梯度的平滑程度的参数
- **spiking** –是否输出脉冲，默认为 `True`，在前向传播时使用 `heaviside` 而在反向传播使用替代梯度。若为 `False` 则不使用替代梯度，前向传播时，使用反向传播时的梯度替代函数对应的原函数

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

$$g(x) = \begin{cases} 0, & x < -\frac{1}{\alpha} \\ -\frac{1}{2}\alpha^2|x|x + \alpha x + \frac{1}{2}, & |x| \leq \frac{1}{\alpha} \\ 1, & x > \frac{1}{\alpha} \end{cases}$$



该函数在文章[2471113] 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 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

$$g(x) = \begin{cases} 0, & x < -\frac{1}{\alpha} \\ -\frac{1}{2}\alpha^2|x|x + \alpha x + \frac{1}{2}, & |x| \leq \frac{1}{\alpha} \\ 1, & x > \frac{1}{\alpha} \end{cases}$$

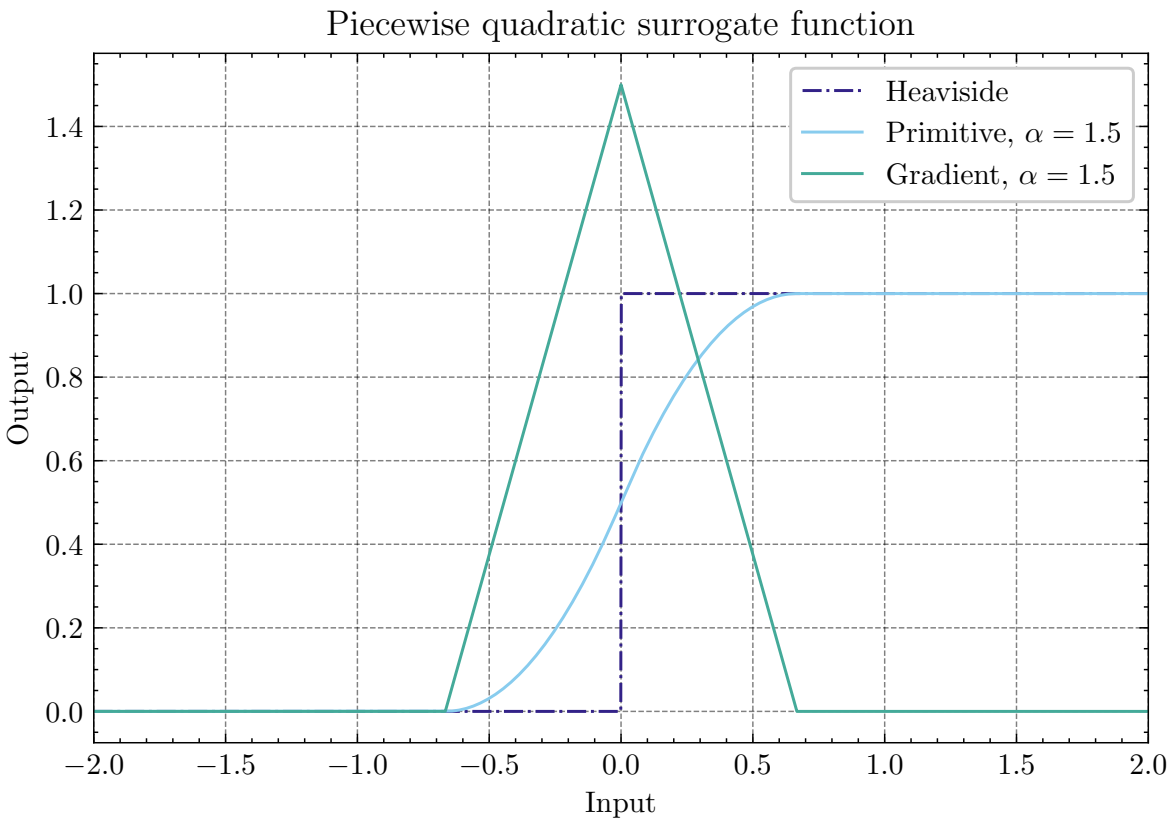
[2] Esser S K, Merolla P A, Arthur J V, et al. Convolutional networks for fast, energy-efficient neuromorphic computing[J]. Proceedings of the national academy of sciences, 2016, 113(41): 11441-11446.

[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.



The function is used in[Page 411, 2][Page 411, 4][Page 411, 7][Page 411, 11][Page 411, 13].

static spiking_function(*x*, *alpha*)

static primitive_function(*x: Tensor*, *alpha*)

training: bool

class spikingjelly.clock_driven.surrogate.**piecewise_exp**

基类：Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**PiecewiseExp**(*alpha=1.0*, *spiking=True*)

基类：*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

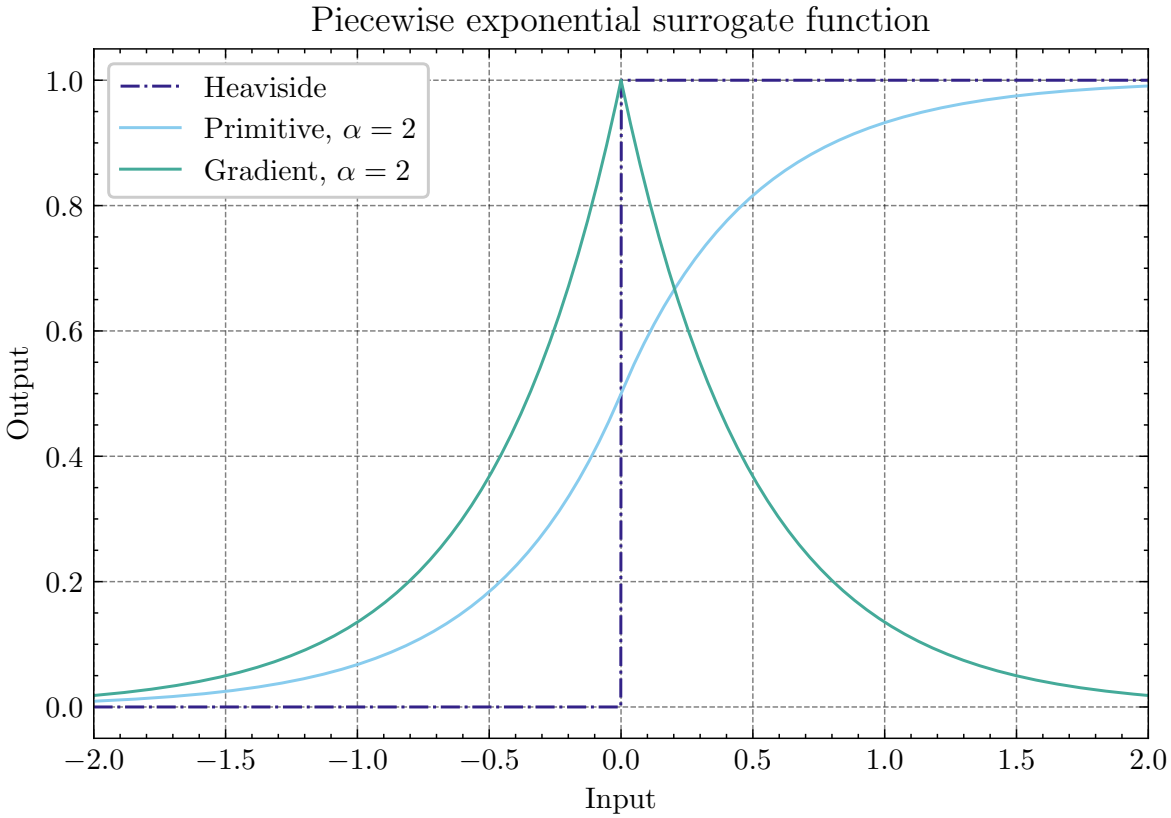
的梯度替代函数对应的原函数

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[6Page 411, 11] 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

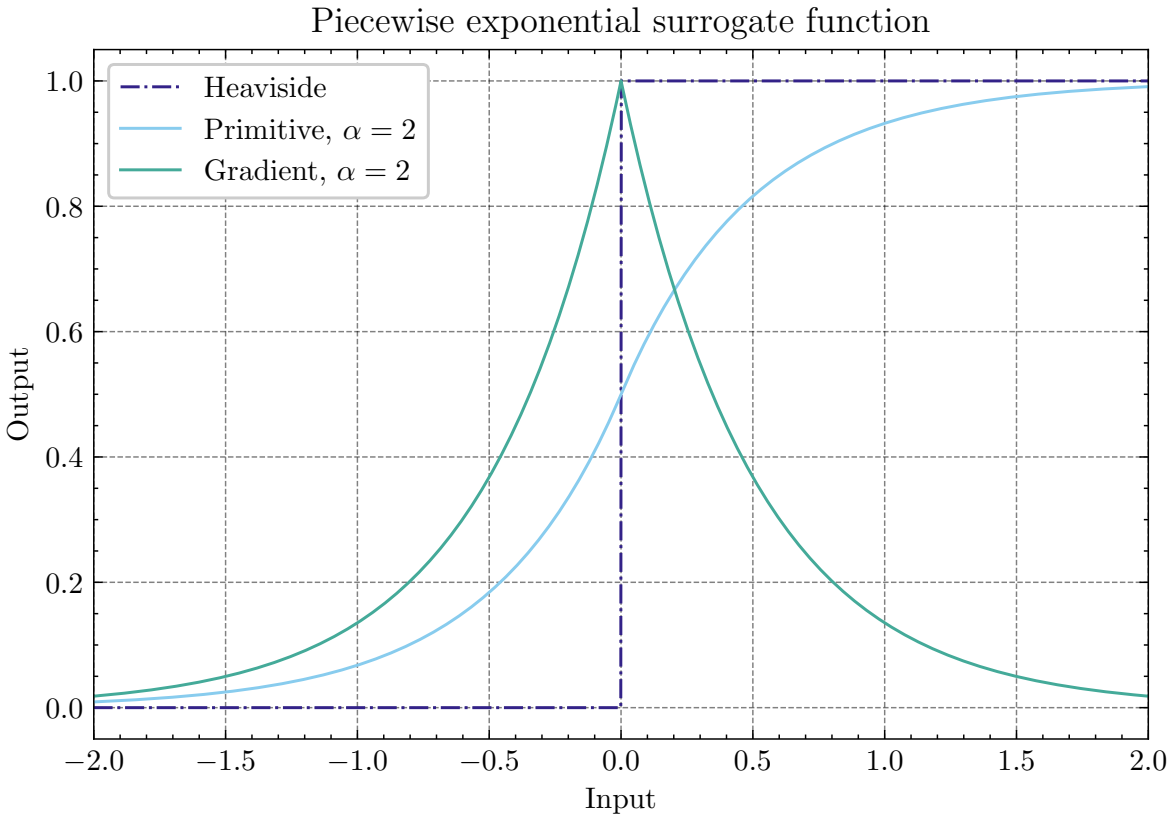
[6] Shrestha S B, Orchard G. SLAYER: spike layer error reassignment in time[C]//Proceedings of the 32nd International Conference on Neural Information Processing Systems. 2018: 1419-1428.

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[6Page 411, 11] .

static spiking_function(*x*, *alpha*)

static primitive_function(*x: Tensor*, *alpha*)

training: bool

class spikingjelly.clock_driven.surrogate.**sigmoid**

基类: Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**Sigmoid**(*alpha=4.0*, *spiking=True*)

基类：*SurrogateFunctionBase*

- *API in English*

参数

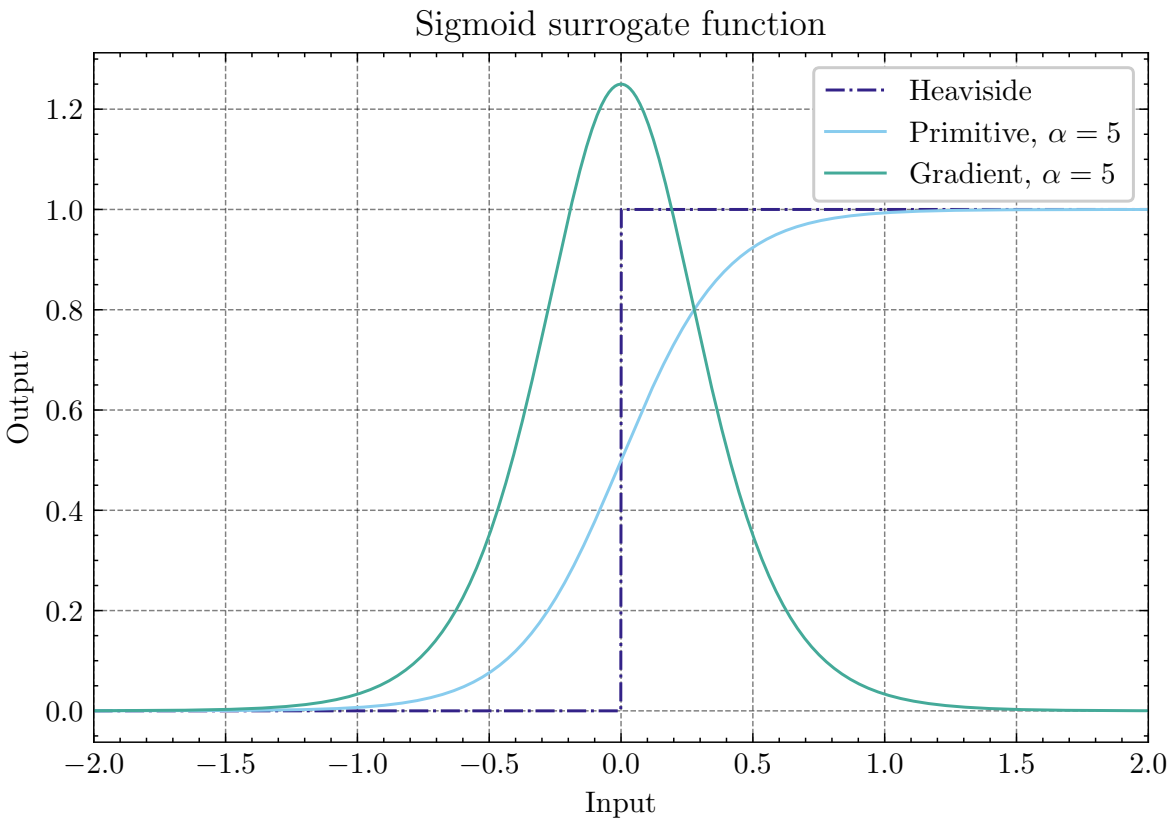
- **alpha** –控制反向传播时梯度的平滑程度的参数
- **spiking** –是否输出脉冲，默认为 `True`，在前向传播时使用 `heaviside` 而在反向传播使用替代梯度。若为 `False` 则不使用替代梯度，前向传播时，使用反向传播时的梯度替代函数对应的原函数

反向传播时使用 sigmoid 的梯度的脉冲发放函数。反向传播为

$$g'(x) = \alpha * (1 - \text{sigmoid}(\alpha x))\text{sigmoid}(\alpha x)$$

对应的原函数为

$$g(x) = \text{sigmoid}(\alpha x) = \frac{1}{1 + e^{-\alpha x}}$$



该函数在文章[Page 411, 4121415] 中使用。

- 中文 *API*

参数

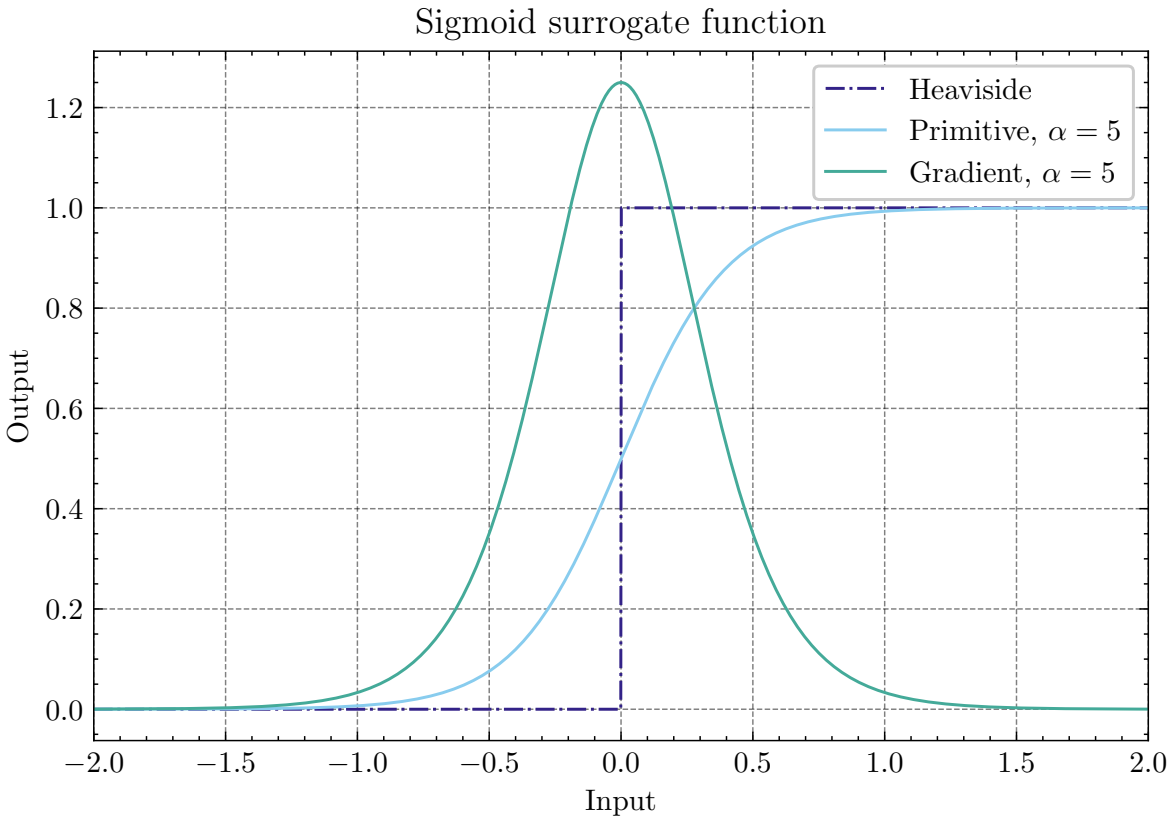
- **alpha** –parameter to control smoothness of gradient
- **spiking** –whether output spikes. The default is `True` which means that using `heaviside` in forward propagation and using surrogate gradient in backward propagation. If `False`, in forward propagation, using the primitive function of the surrogate gradient function used in backward propagation

The sigmoid surrogate spiking function. The gradient is defined by

$$g'(x) = \alpha * (1 - \text{sigmoid}(\alpha x))\text{sigmoid}(\alpha x)$$

The primitive function is defined by

$$g(x) = \text{sigmoid}(\alpha x) = \frac{1}{1 + e^{-\alpha x}}$$



[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 function is used in[Page 411, 4][Page 416, 12][Page 416, 14][Page 416, 15] .

static spiking_function(*x*, *alpha*)

static primitive_function(*x: Tensor*, *alpha*)

cuda_code(*x: str*, *y: str*, *dtype='fp32'*)

training: bool

class spikingjelly.clock_driven.surrogate.**soft_sign**

基类：Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**SoftSign**(*alpha=2.0*, *spiking=True*)

基类：*SurrogateFunctionBase*

- *API in English*

参数

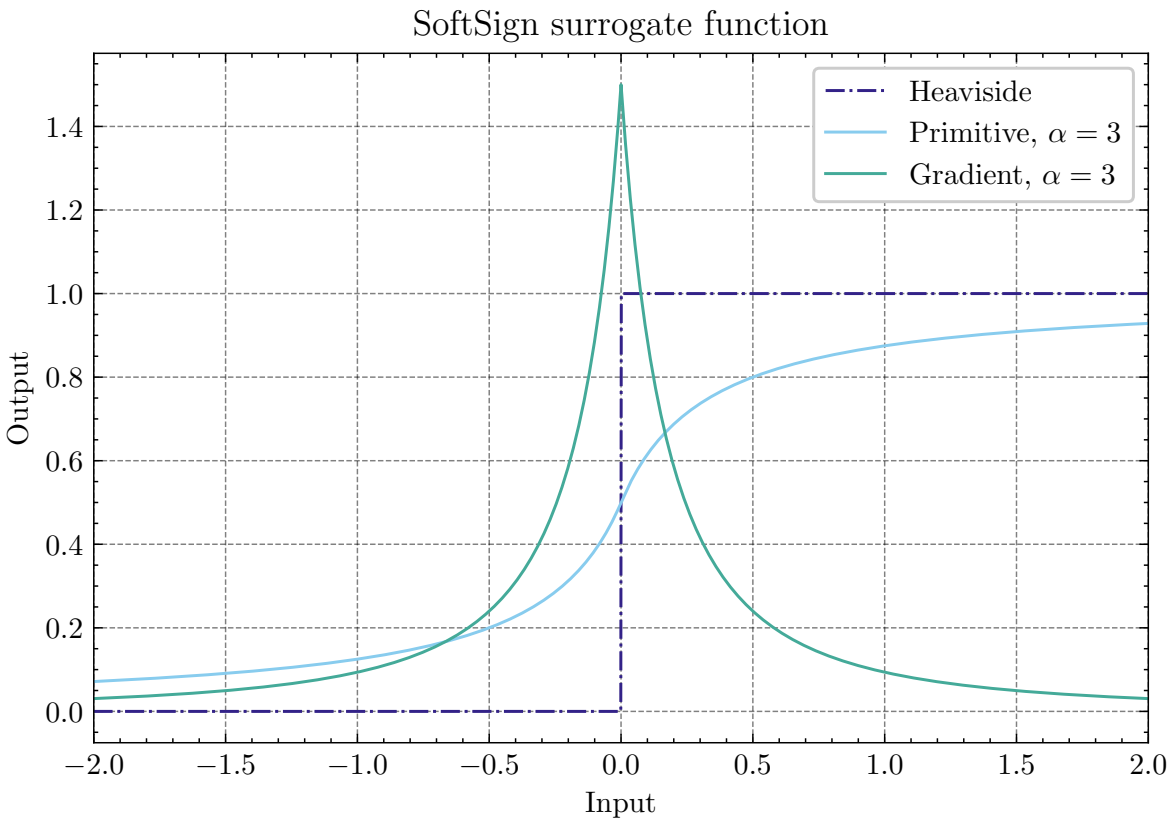
- **alpha** –控制反向传播时梯度的平滑程度的参数
- **spiking** –是否输出脉冲，默认为 True，在前向传播时使用 heaviside 而在反向传播使用替代梯度。若为 False 则不使用替代梯度，前向传播时，使用反向传播时的梯度替代函数对应的原函数

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 Page 411, 11] 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

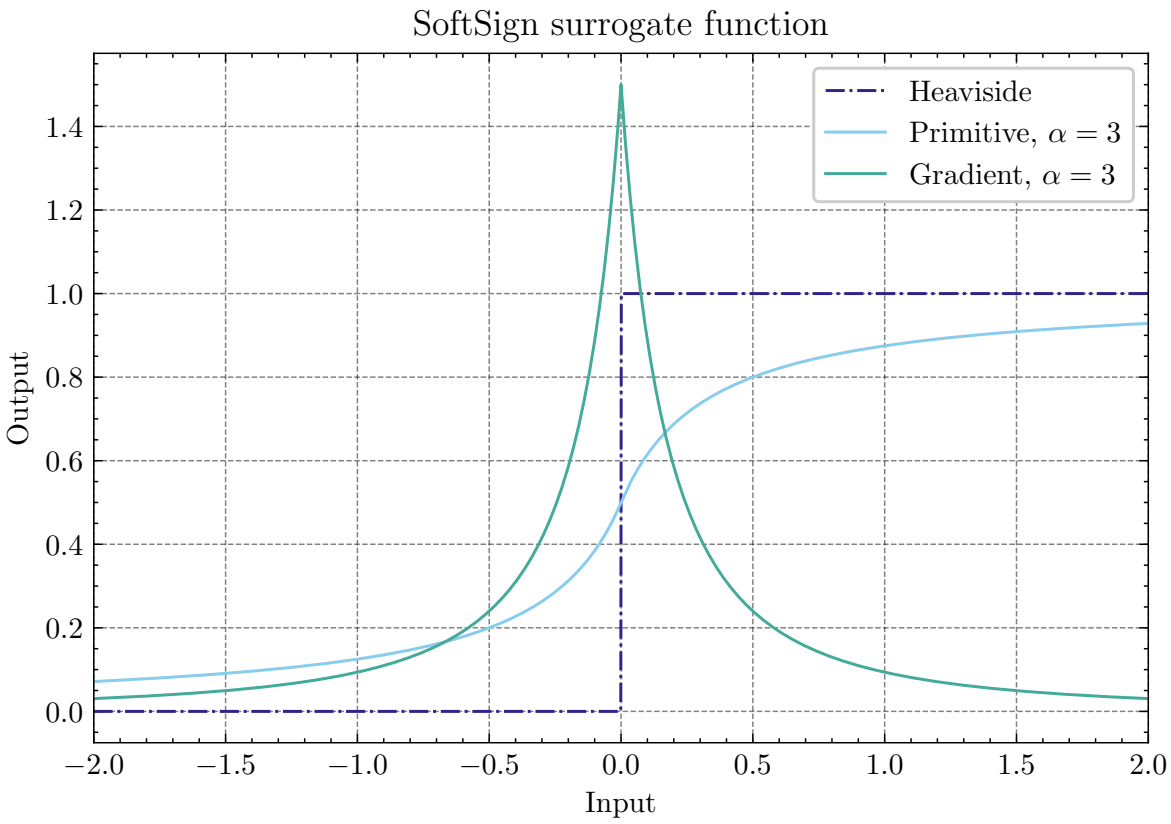
The soft sign surrogate spiking function. The gradient is defined by

$$g'(x) = \frac{\alpha}{2(1 + |\alpha x|)^2}$$

The primitive function is defined by

$$g(x) = \frac{1}{2}(\frac{\alpha x}{1 + |\alpha x|} + 1)$$

[8] Zenke F, Ganguli S. Superspike: Supervised learning in multilayer spiking neural networks[J]. Neural computation, 2018, 30(6): 1514-1541.



The function is used in[8Page 411, 11] .

static spiking_function(*x*, *alpha*)

static primitive_function(*x: Tensor*, *alpha*)

training: bool

class spikingjelly.clock_driven.surrogate.**atan**

基类：Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**ATan**(*alpha=2.0*, *spiking=True*)

基类：*SurrogateFunctionBase*

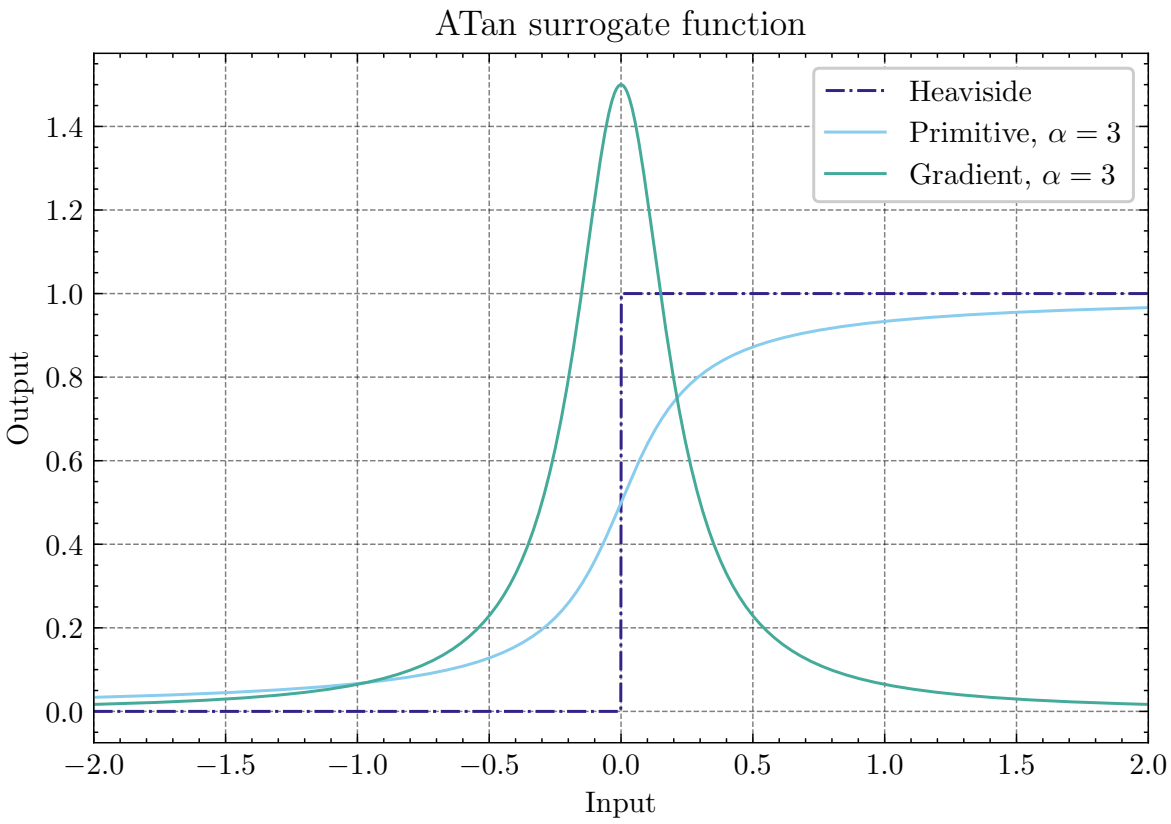
- *API in English*

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

$$g(x) = \frac{1}{\pi}\arctan(\frac{\pi}{2}\alpha x) + \frac{1}{2}$$



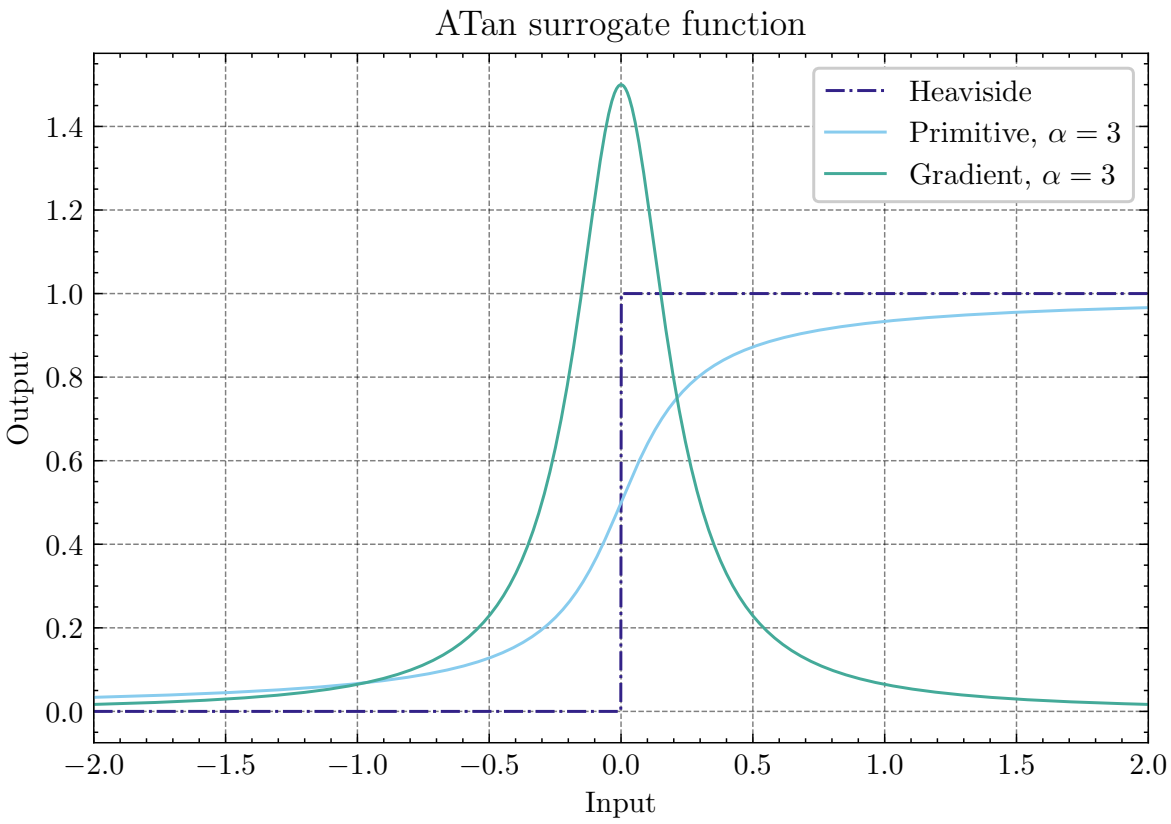
- 中文 *API*

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: Tensor*, *alpha*)

cuda_code(*x: str*, *y: str*, *dtype='fp32'*)

training: bool

class spikingjelly.clock_driven.surrogate.**nonzero_sign_log_abs**

基类：Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**NonzeroSignLogAbs**(*alpha=1.0*, *spiking=True*)

基类：*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

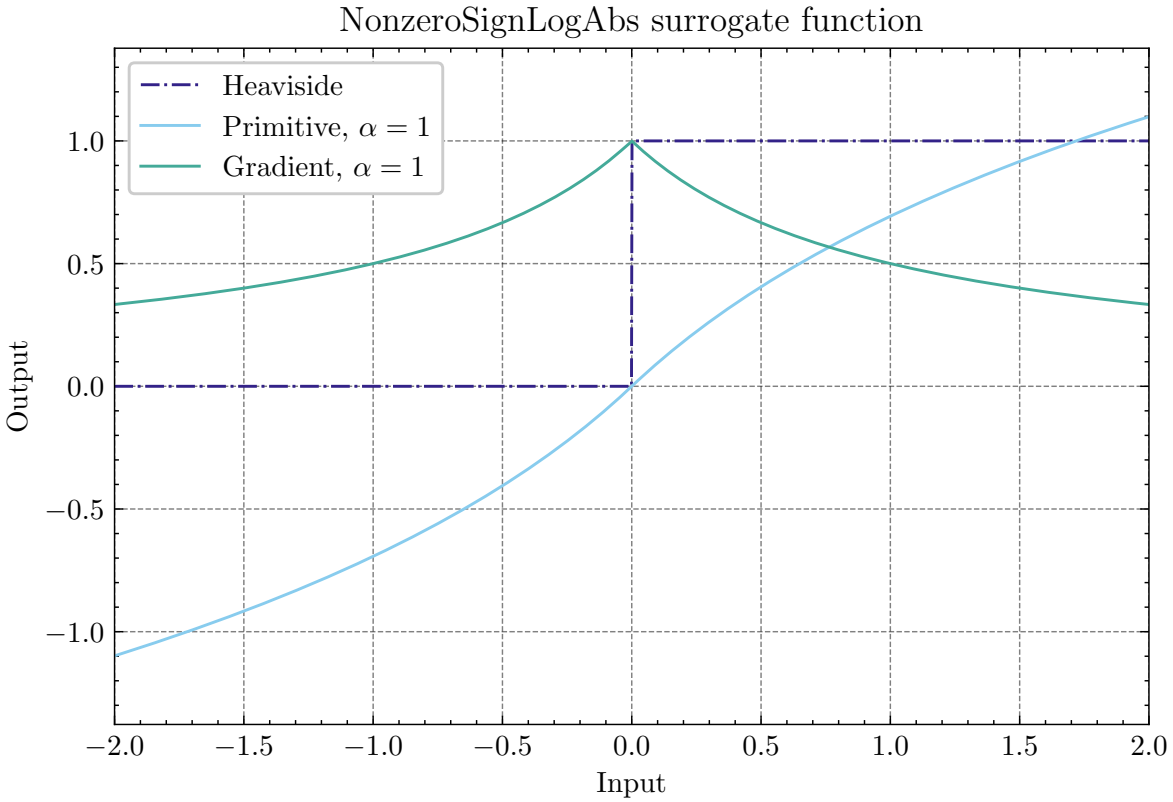
$$g'(x) = \frac{\alpha}{1 + |\alpha x|} = \frac{1}{\frac{1}{\alpha} + |x|}$$

对应的原函数为

$$g(x) = \text{NonzeroSign}(x) \log(|\alpha x| + 1)$$

其中

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



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

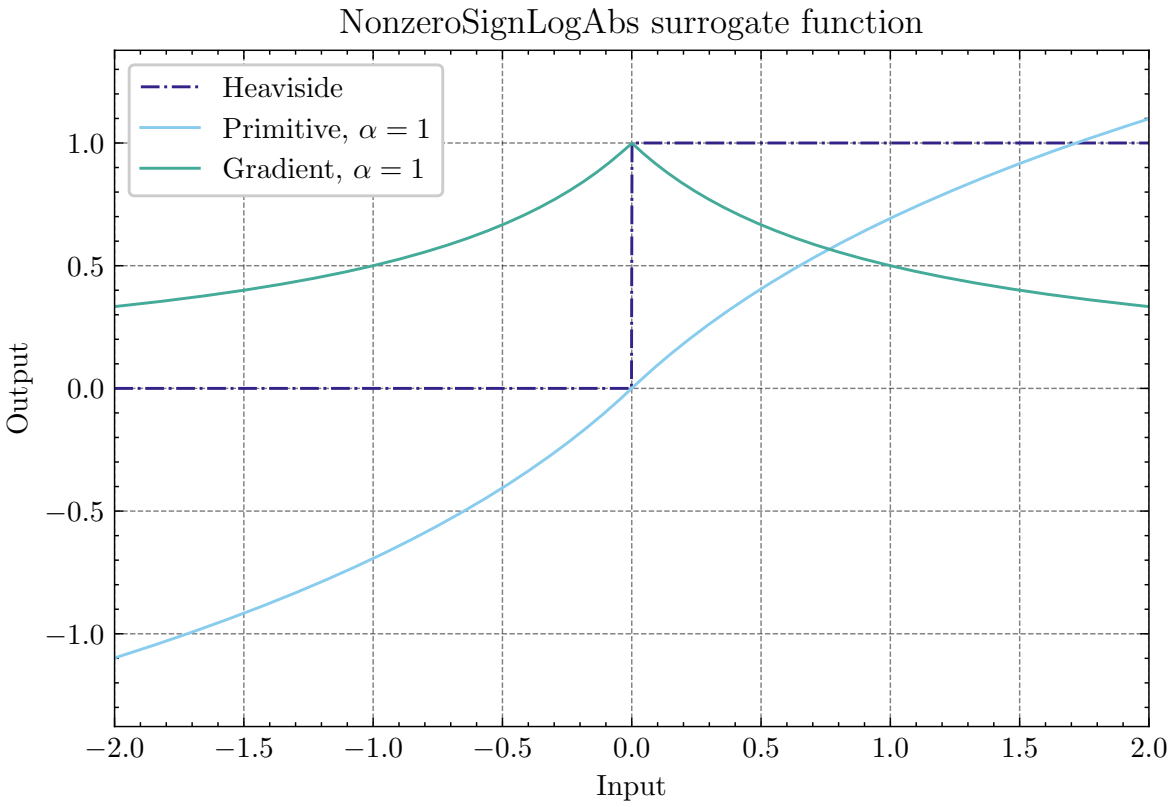
$$g'(x) = \frac{\alpha}{1 + |\alpha x|} = \frac{1}{\frac{1}{\alpha} + |x|}$$

The primitive function is defined by

$$g(x) = \text{NonzeroSign}(x) \log(|\alpha x| + 1)$$

where

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



The function is used in .

static spiking_function(*x*, *alpha*)

static primitive_function(*x: Tensor*, *alpha*)

training: bool

class spikingjelly.clock_driven.surrogate.**erf**

基类：Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**Erf**(*alpha=2.0*, *spiking=True*)

基类：*SurrogateFunctionBase*

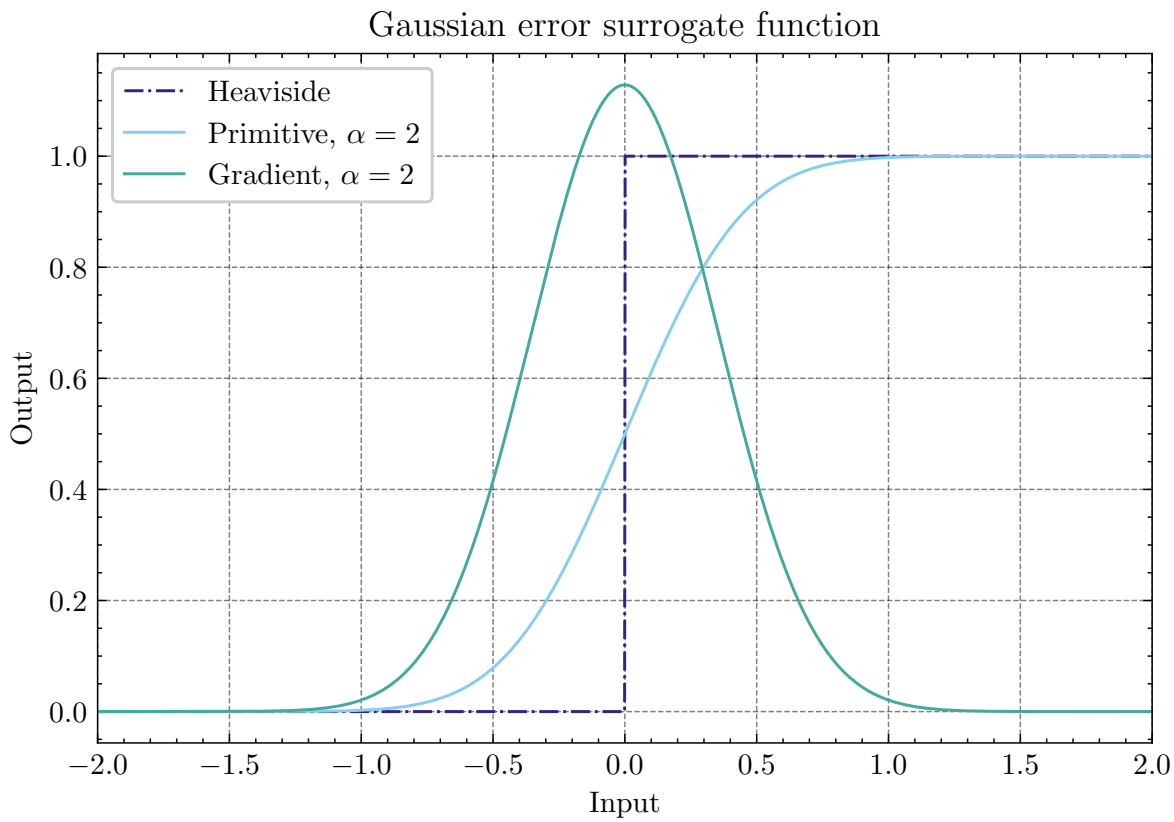
- *API in English*

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 Page 411, 418] 中使用。

- 中文 *API*

参数

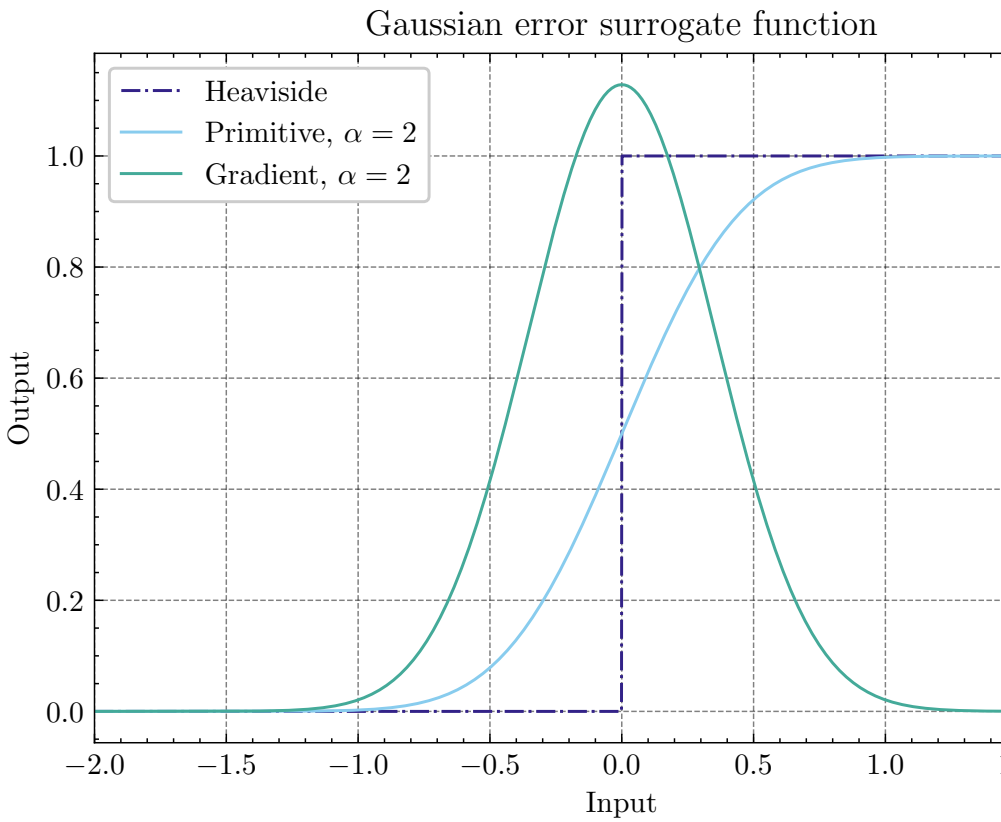
- **alpha** –parameter to control smoothness of gradient
- **spiking** –whether output spikes. The default is `True` which means that using `heaviside` in forward propagation and using surrogate gradient in backward propagation. If `False`, in forward propagation, using the primitive function of the 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[1 Page 411, 4 Page 425, 18].

static spiking_function(*x*, *alpha*)

static primitive_function(*x: Tensor*, *alpha*)

training: bool

class spikingjelly.clock_driven.surrogate.**piecewise_leaky_relu**

基类：Function

static forward(*ctx*, *x: Tensor*, *w=1*, *c=0.01*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**PiecewiseLeakyReLU**(*w=1.0*, *c=0.01*, *spiking=True*)

基类：*MultiArgsSurrogateFunctionBase*

- *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`

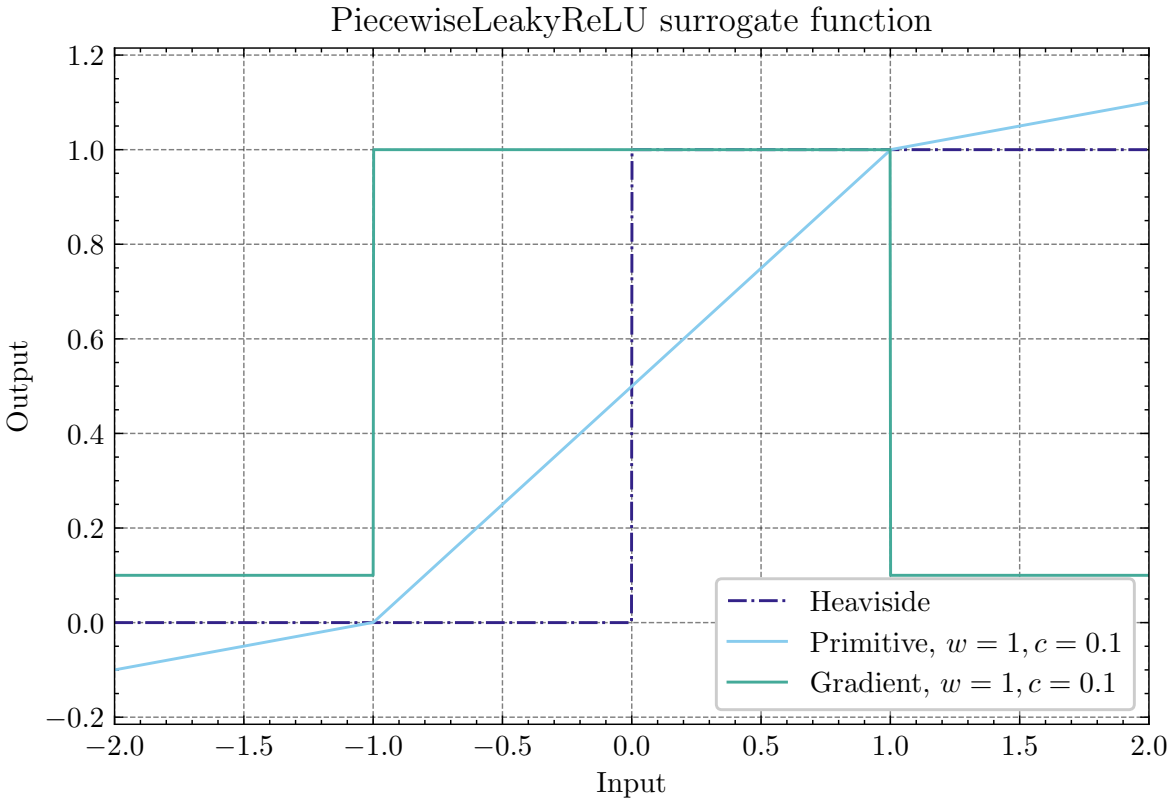
- **spiking** –是否输出脉冲，默认为 `True`，在前向传播时使用 `heaviside` 而在反向传播使用替代梯度。若为 `False` 则不使用替代梯度，前向传播时，使用反向传播时的梯度替代函数对应的原函数

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[3Page 411, 45910Page 416, 121617] 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.

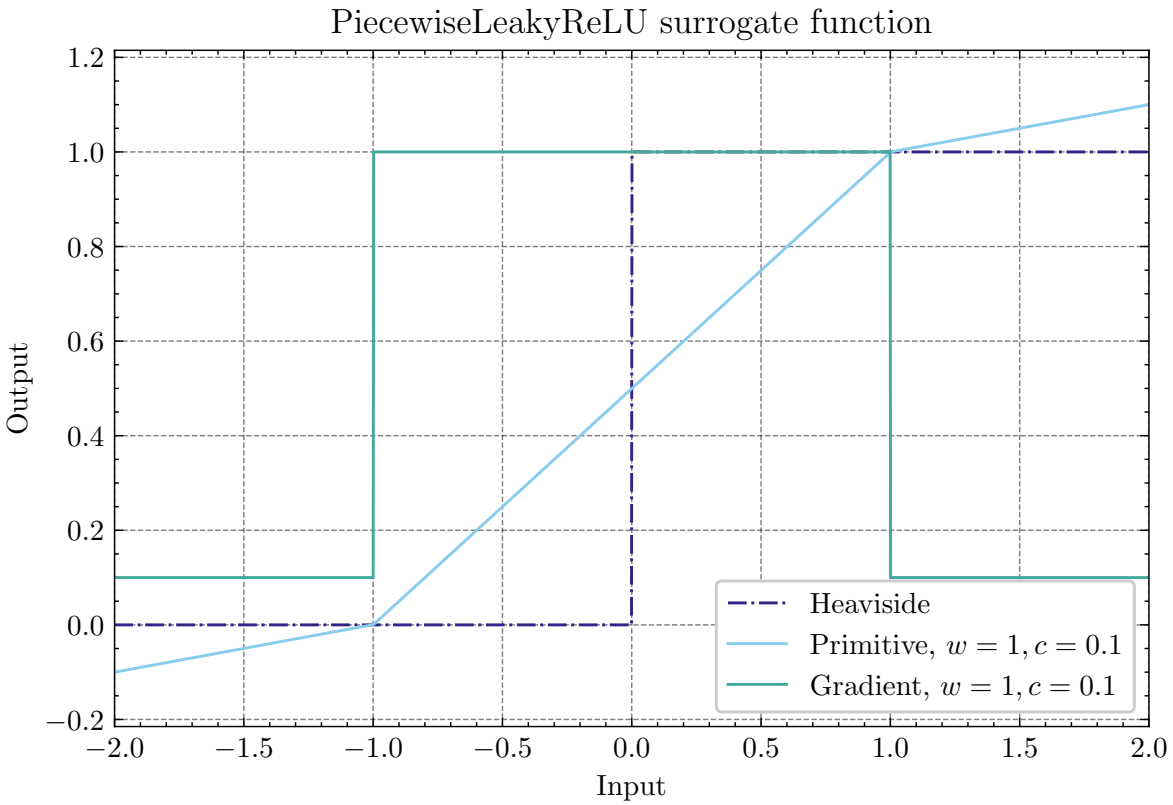
[5] Huh D, Sejnowski T J. Gradient descent for spiking neural networks[C]//Proceedings of the 32nd International Conference on Neural Information Processing Systems. 2018: 1440-1450.

[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[3Page 411, 4Page 428, 5Page 428, 9Page 428, 10Page 416, 12Page 428, 16Page 428, 17].

forward(*x*)

static spiking_function(*x: Tensor*, *w*, *c*)

static primitive_function(*x: Tensor*, *w*, *c*)

cuda_code(*x: str*, *y: str*, *dtype='fp32'*)

training: bool

class spikingjelly.clock_driven.surrogate.**squarewave_fourier_series**

基类：Function

static forward(*ctx*, *x: Tensor*, *n: int*, *T_period: float*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**SquarewaveFourierSeries**(*n: int = 2*, *T_period: float = 8*, *spiking=True*)

基类：*MultiArgsSurrogateFunctionBase*

forward(*x*)

static spiking_function(*x: Tensor*, *w*, *c*)

static primitive_function(*x: Tensor*, *n: int*, *T_period: float*)

cuda_code(*x: str*, *y: str*, *dtype='fp32'*)

training: bool

class spikingjelly.clock_driven.surrogate.**s2nn**

基类：Function

static forward(*ctx*, *x: Tensor*, *alpha: float*, *beta: float*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**S2NN**(*alpha=4.0*, *beta=1.0*, *spiking=True*)

基类：*MultiArgsSurrogateFunctionBase*

- *API in English*

参数

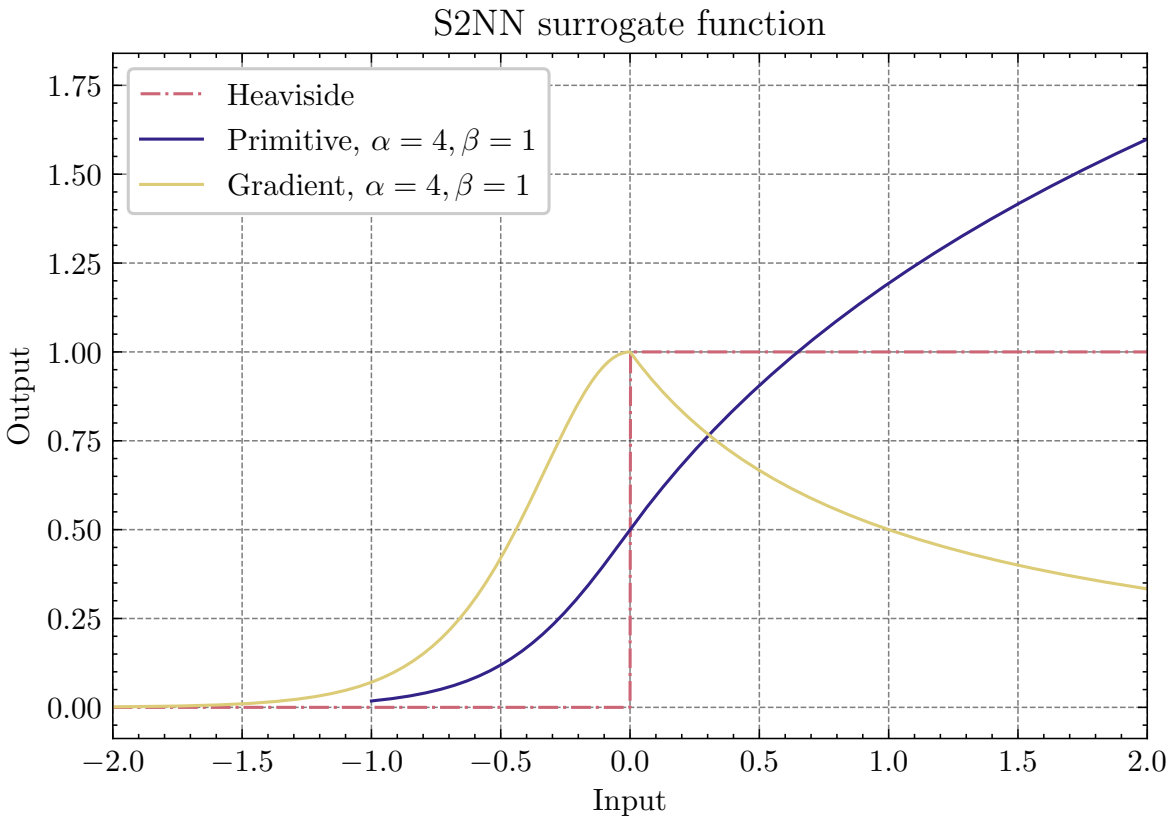
- **alpha** –控制 x < 0 时梯度的参数
- **beta** –控制 x >= 0 时梯度的参数
- **spiking** –是否输出脉冲，默认为 True，在前向传播时使用 heaviside 而在反向传播使用替代梯度。若为 False 则不使用替代梯度，前向传播时，使用反向传播时的梯度替代函数对应的原函数

S2NN: Time Step Reduction of Spiking Surrogate Gradients for Training Energy Efficient Single-Step Neural Networks 提出的 S2NN 替代函数。反向传播为

$$g'(x) = \begin{cases} \alpha * (1 - \text{sigmoid}(\alpha x))\text{sigmoid}(\alpha x), x < 0 \\ \beta(x + 1), x \geq 0 \end{cases}$$

对应的原函数为

$$g(x) = \begin{cases} \text{sigmoid}(\alpha x), x < 0 \\ \beta \ln(x + 1) + 1, x \geq 0 \end{cases}$$



- 中文 *API*

参数

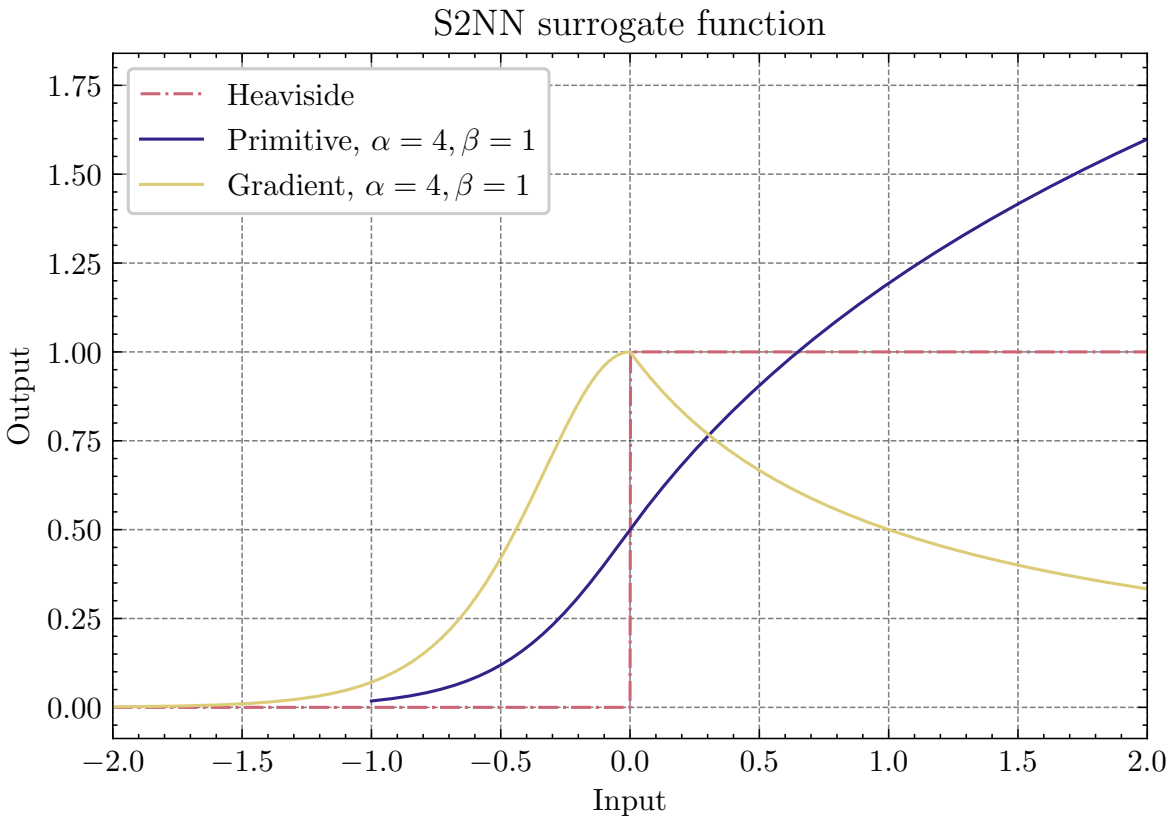
- **alpha** –the param that controls the gradient when `x < 0`
- **beta** –the param that controls the gradient when `x >= 0`
- **spiking** –whether output spikes. The default is `True` which means that using `heaviside` in forward propagation and using surrogate gradient in backward propagation. If `False`, in forward propagation, using the primitive function of the surrogate gradient function used in backward propagation

The S2NN surrogate spiking function, which is proposed by S2NN: Time Step Reduction of Spiking Surrogate Gradients for Training Energy Efficient Single-Step Neural Networks. The gradient is defined by

$$g'(x) = \begin{cases} \alpha * (1 - \text{sigmoid}(\alpha x))\text{sigmoid}(\alpha x), x < 0 \\ \beta(x + 1), x \geq 0 \end{cases}$$

The primitive function is defined by

$$g(x) = \begin{cases} \text{sigmoid}(\alpha x), x < 0 \\ \beta \ln(x + 1) + 1, x \geq 0 \end{cases}$$



forward(*x*)

static spiking_function(*x: Tensor*, *alpha*, *beta*)

static primitive_function(*x: Tensor*, *alpha*, *beta*)

cuda_code(*x: str*, *y: str*, *dtype='fp32'*)

training: bool

class spikingjelly.clock_driven.surrogate.**q_pseudo_spike**

基类: Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class spikingjelly.clock_driven.surrogate.**QPseudoSpike**(*alpha=2.0*, *spiking=True*)

基类: *SurrogateFunctionBase*

- *API in English*

参数

- **alpha** –控制反向传播时梯度函数尾部厚度的参数

- **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

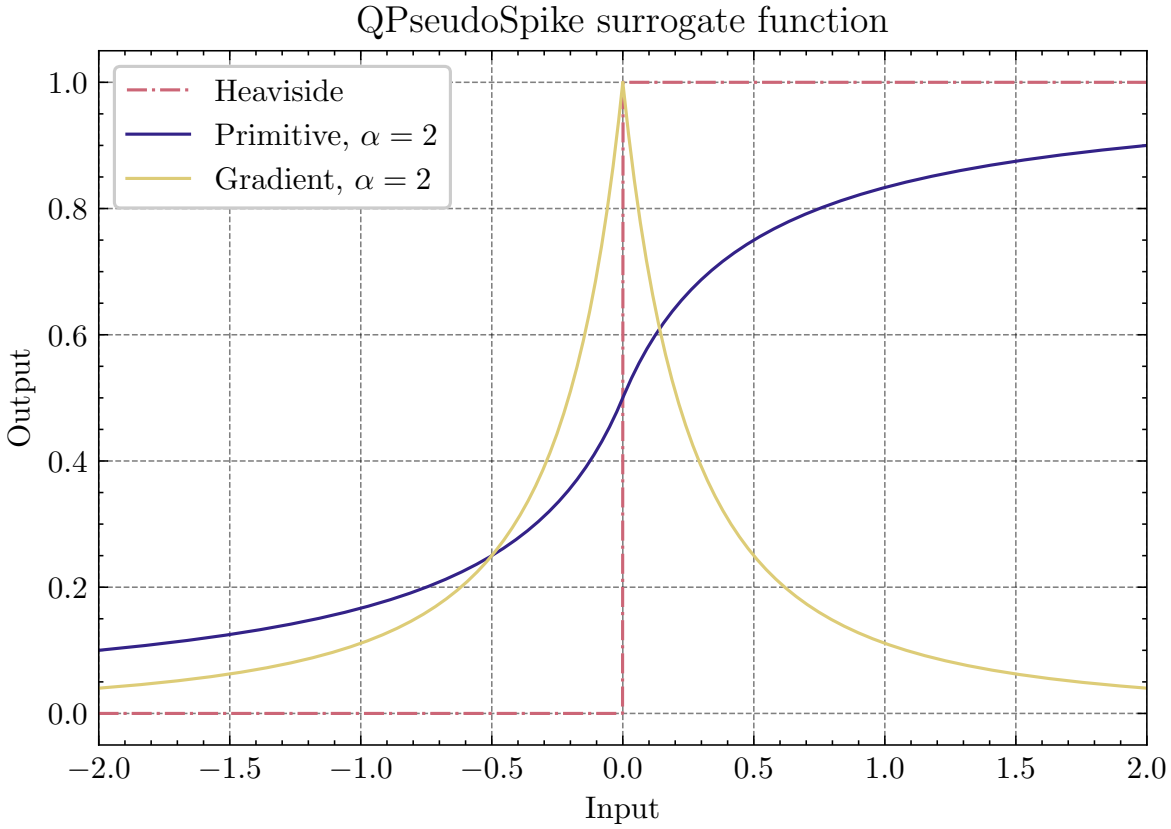
Surrogate Gradients Design 提出的 q-PseudoSpike 替代函数。反向传播为

$$g'(x) = (1 + \frac{2|x|}{\alpha - 1})^{-\alpha}$$

其中 $\alpha > 1$ 对应原文中的 q。

对应的原函数为

$$g(x) = \begin{cases} \frac{1}{2}(1 - \frac{2x}{\alpha-1})^{1-\alpha}, & x < 0 \\ 1 - \frac{1}{2}(1 + \frac{2x}{\alpha-1})^{1-\alpha}, & x \geq 0. \end{cases}$$



- 中文 *API*

参数

- **alpha** –parameter to control tail fatness 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

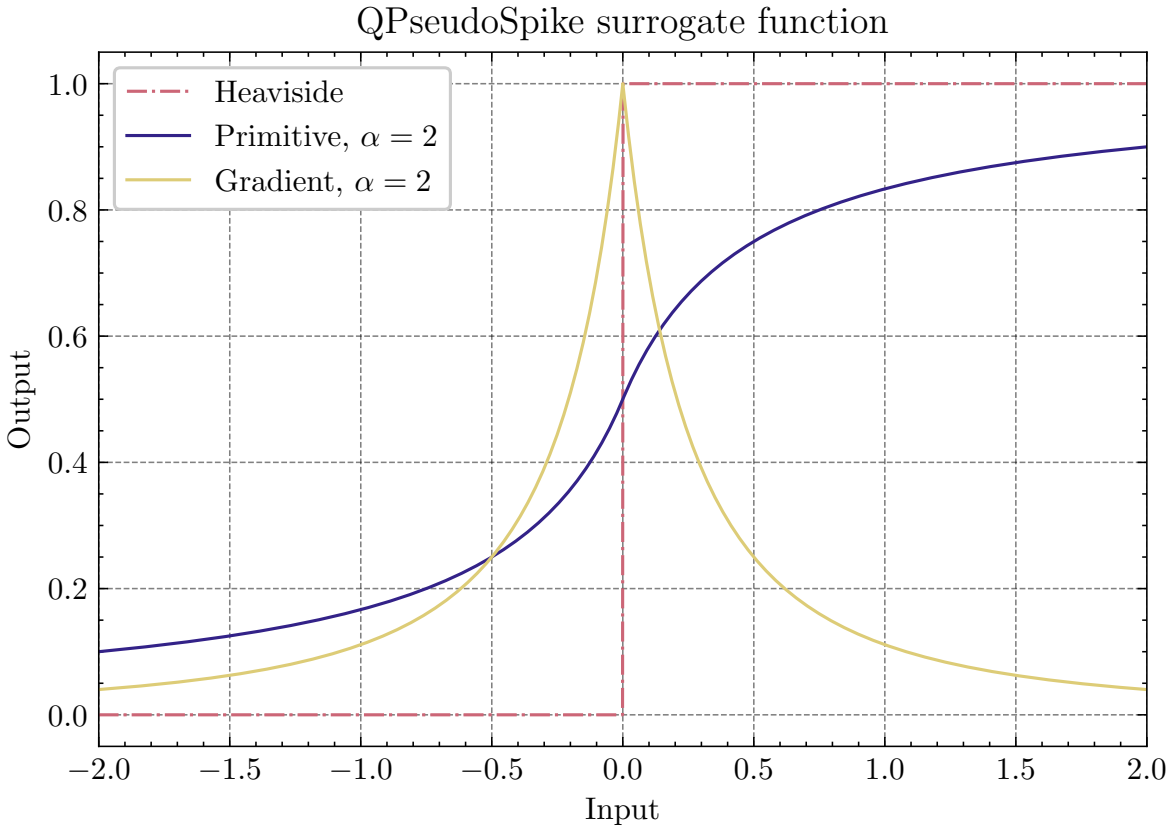
The q-PseudoSpike surrogate spiking function, which is first proposed in Surrogate Gradients Design. The gradient is defined by

$$g'(x) = (1 + \frac{2|x|}{\alpha - 1})^{-\alpha}$$

where $\alpha > 1$ corresponds to q in paper.

The primitive function is defined by

$$g(x) = \begin{cases} \frac{1}{2}(1 - \frac{2x}{\alpha-1})^{1-\alpha}, & x < 0 \\ 1 - \frac{1}{2}(1 + \frac{2x}{\alpha-1})^{1-\alpha}, & x \geq 0. \end{cases}$$



static spiking_function(*x*, *alpha*)

static primitive_function(*x: Tensor*, *alpha*)

training: bool

cuda_code(*x: str*, *y: str*, *dtype='fp32'*)

References

spikingjelly.clock_driven.ann2snn package

Subpackages

spikingjelly.clock_driven.ann2snn.examples package

Submodules

spikingjelly.clock_driven.ann2snn.examples.if_cnn_mnist module

spikingjelly.clock_driven.ann2snn.examples.cnn_mnist.**val**(*net*, *device*, *data_loader*, *T=None*)

spikingjelly.clock_driven.ann2snn.examples.cnn_mnist.**main**()

Module contents

spikingjelly.clock_driven.ann2snn.kernels package

Submodules

spikingjelly.clock_driven.ann2snn.kernels.onnx module

spikingjelly.clock_driven.ann2snn.kernels.pytorch module

Module contents

Submodules

spikingjelly.clock_driven.ann2snn.modules module

class spikingjelly.clock_driven.ann2snn.modules.**VoltageHook**(*scale=1.0*, *momentum=0.1*, *mode='Max'*)

基类：Module

- *API in English*

参数

- **scale**(*float*) –缩放初始值

- **momentum** (*float*) –动量值
- **mode** (*str, float*) –模式。输入“Max”表示记录 ANN 激活最大值，“99.9%”表示记录 ANN 激活的 99.9% 分位点，输入 0-1 的 float 型浮点数表示记录激活最大值的对应倍数。

VoltageHook 用于在 ANN 推理中确定激活的范围。

- 中文 *API*

参数

- **scale** (*float*) –initial scaling value
- **momentum** (*float*) –momentum value
- **mode** (*str, float*) –The mode. Value “Max” means recording the maximum value of ANN activation, “99.9%” means recording the 99.9% precentile of ANN activation, and a float of 0-1 means recording the corresponding multiple of the maximum activation value.

VoltageHook is used to determine the range of activations in ANN inference.

forward(*x*)

training: bool

class spikingjelly.clock_driven.ann2snn.modules.**VoltageScaler**(*scale=1.0*)

基类：Module

- *API in English*

参数

scale (*float*) –缩放值

VoltageScaler 用于 SNN 推理中缩放电流。

- 中文 *API*

参数

scale (*float*) –scaling value

VoltageScaler is used for scaling current in SNN inference.

forward(*x*)

extra_repr()

training: bool

Module contents

spikingjelly.clock_driven.lava_exchange package

Module contents

spikingjelly.clock_driven.lava_exchange.**TNX_to_NXT**(*x_seq: Tensor*)

spikingjelly.clock_driven.lava_exchange.**NXT_to_TNX**(*x_seq: Tensor*)

spikingjelly.clock_driven.lava_exchange.**lava_neuron_forward**(*lava_neuron: Module*, *x_seq: Tensor*, *v: Tensor*)

spikingjelly.clock_driven.lava_exchange.**step_quantize**(*x: Tensor*, *step: float = 1.0*)

参数

- **x** (*torch.Tensor*) –the input tensor
- **step** (*float*) –the quantize step

返回

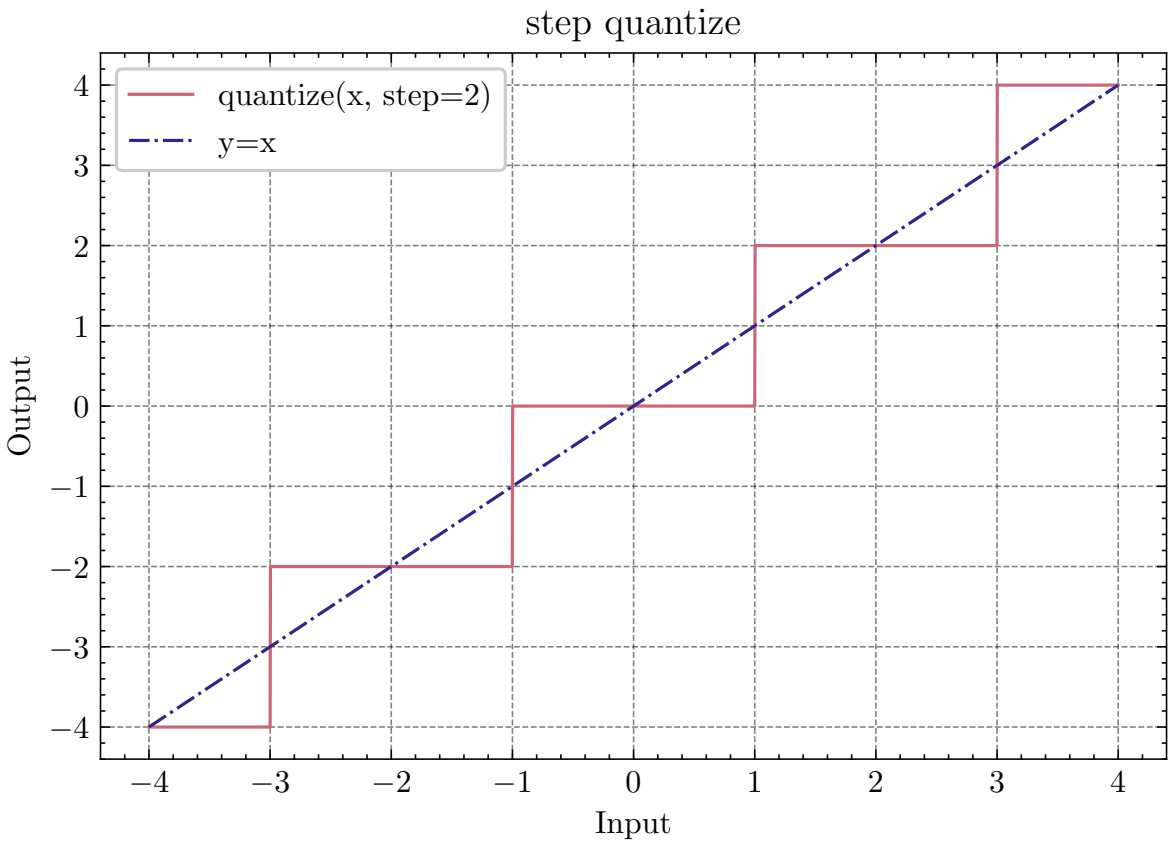
quantized tensor

返回类型

torch.Tensor

The step quantize function. Here is an example:

```
# plt.style.use(['science', 'muted', 'grid'])
fig = plt.figure(dpi=200, figsize=(6, 4))
x = torch.arange(-4, 4, 0.001)
plt.plot(x, lava_exchange.step_quantize(x, 2.), label='quantize(x, step=2)')
plt.plot(x, x, label='y=x', ls='-.')
plt.legend()
plt.grid(ls='--')
plt.title('step quantize')
plt.xlabel('Input')
plt.ylabel('Output')
plt.savefig('./docs/source/_static/API/clock_driven/lava_exchange/step_quantize.
↪svg')
plt.savefig('./docs/source/_static/API/clock_driven/lava_exchange/step_quantize.
↪pdf')
```



spikingjelly.clock_driven.lava_exchange.**quantize_8bit**(*x: Tensor*, *scale*, *descale=False*)

spikingjelly.clock_driven.lava_exchange.**check_conv2d**(*conv2d_nn: Conv2d*)

spikingjelly.clock_driven.lava_exchange.**check_fc**(*fc: Linear*)

spikingjelly.clock_driven.lava_exchange.**to_lava_neuron_param_dict**(*sj_ms_neuron: Module*)

spikingjelly.clock_driven.lava_exchange.**to_lava_neuron**(*sj_ms_neuron: Module*)

spikingjelly.clock_driven.lava_exchange.**linear_to_lava_synapse_dense**(*fc: Linear*)

参数

fc (*nn.Linear*) –a pytorch linear layer without bias

返回

a lava slayer dense synapse

返回类型

slayer.synapse.Dense

Codes example:

```
T = 4
N = 2
layer_nn = nn.Linear(8, 4, bias=False)
layer_sl = lava_exchange.linear_to_lava_synapse_dense(layer_nn)
x_seq = torch.rand([T, N, 8])
with torch.no_grad():
    y_nn = functional.seq_to_ann_forward(x_seq, layer_nn)
    y_sl = lava_exchange.NXT_to_TNX(layer_sl(lava_exchange.TNX_to_NXT(x_seq)))
    print('max error:', (y_nn - y_sl).abs().max())
```

spikingjelly.clock_driven.lava_exchange.**conv2d_to_lava_synapse_conv**(*conv2d_nn: Conv2d*)

参数

conv2d_nn (*nn.Conv2d*) –a pytorch conv2d layer without bias

返回

a lava slayer conv synapse

返回类型

slayer.synapse.Conv

Codes example:

```
T = 4
N = 2
layer_nn = nn.Conv2d(3, 8, kernel_size=3, stride=1, padding=1, bias=False)
layer_sl = lava_exchange.conv2d_to_lava_synapse_conv(layer_nn)
x_seq = torch.rand([T, N, 3, 28, 28])
with torch.no_grad():
    y_nn = functional.seq_to_ann_forward(x_seq, layer_nn)
    y_sl = lava_exchange.NXT_to_TNX(layer_sl(lava_exchange.TNX_to_NXT(x_seq)))
    print('max error:', (y_nn - y_sl).abs().max())
```

spikingjelly.clock_driven.lava_exchange.**avgpool2d_to_lava_synapse_pool**(*pool2d_nn: AvgPool2d*)

参数

pool2d_nn (*nn.AvgPool2d*) –a pytorch AvgPool2d layer

返回

a lava slayer pool layer

返回类型

slayer.synapse.Pool

Warning

The lava slayer pool layer applies sum pooling, rather than average pooling.

```
T = 4
N = 2
layer_nn = nn.AvgPool2d(kernel_size=2, stride=2)
layer_sl = lava_exchange.avgpool2d_to_lava_synapse_pool(layer_nn)
x_seq = torch.rand([T, N, 3, 28, 28])
with torch.no_grad():
    y_nn = functional.seq_to_ann_forward(x_seq, layer_nn)
    y_sl = lava_exchange.NXT_to_TNX(layer_sl(lava_exchange.TNX_to_NXT(x_seq))) /␣
↪4.
    print('max error:', (y_nn - y_sl).abs().max())
```

spikingjelly.clock_driven.lava_exchange.**to_lava_block_dense**(*fc: Linear*, *sj_ms_neuron: Module*, *quantize_to_8bit: bool = True*)

spikingjelly.clock_driven.lava_exchange.**to_lava_block_conv**(*conv2d_nn: Conv2d*, *sj_ms_neuron: Module*, *quantize_to_8bit: bool = True*)

spikingjelly.clock_driven.lava_exchange.**to_lava_block_flatten**(*flatten_nn: Flatten*)

Module contents

7.5.2 spikingjelly.datasets package

Submodules

spikingjelly.datasets.asl_dvs module

class spikingjelly.datasets.asl_dvs.**ASLDVS**(*root: str*, *data_type: str = 'event'*, *frames_number: Optional[int] = None*, *split_by: Optional[str] = None*, *duration: Optional[int] = None*, *custom_integrate_function: Optional[Callable] = None*, *custom_integrated_frames_dir_name: Optional[str] = None*, *transform: Optional[Callable] = None*, *target_transform: Optional[Callable] = None*)

基类：*NeuromorphicDatasetFolder*

The ASL-DVS dataset, which is proposed by Graph-based Object Classification for Neuromorphic Vision Sensing.

Refer to *spikingjelly.datasets.NeuromorphicDatasetFolder* for more details about params information.

static resource_url_md5() → list

返回

A list `url` that `url[i]` is a tuple, which contains the i-th file's name, download link, and MD5

返回类型

list

static downloadable() → bool

返回

Whether the dataset can be directly downloaded by python codes. If not, the user have to download it manually

返回类型

bool

static extract_downloaded_files(*download_root: str*, *extract_root: str*)

参数

- **download_root** (*str*) –Root directory path which saves downloaded dataset files
- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files

返回

None

This function defines how to extract download files.

static load_origin_data(*file_name: str*) → Dict

参数

file_name (*str*) –path of the events file

返回

a dict whose keys are `['t', 'x', 'y', 'p']` and values are `numpy.ndarray`

返回类型

Dict

This function defines how to read the origin binary data.

static get_H_W() → Tuple

返回

A tuple `(H, W)`, where `H` is the height of the data and `W` is the weight of the data. For example, this function returns `(128, 128)` for the DVS128 Gesture dataset.

返回类型

tuple

static read_mat_save_to_np(*mat_file: str*, *np_file: str*)

static create_events_np_files(*extract_root: str*, *events_np_root: str*)

参数

- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files
- **events_np_root** –Root directory path which saves events files in the `npz` format

返回

None

This function defines how to convert the origin binary data in `extract_root` to `npz` format and save converted files in `events_np_root`.

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*, *data_type: str = 'event'*, *frames_number: Optional[int] = None*, *split_by: Optional[str] = None*, *duration: Optional[int] = None*, *custom_integrate_function: Optional[Callable] = None*, *custom_integrated_frames_dir_name: Optional[str] = None*, *transform: Optional[Callable] = None*, *target_transform: Optional[Callable] = None*)

基类：*NeuromorphicDatasetFolder*

The CIFAR10-DVS dataset, which is proposed by 'CIFAR10-DVS: An Event-Stream Dataset for Object Classification

<https://internal-journal.frontiersin.org/articles/10.3389/fnins.2017.00309/full>'_.

Refer to *spikingjelly.datasets.NeuromorphicDatasetFolder* for more details about params information.

static resource_url_md5() → list

返回

A list `url` that `url[i]` is a tuple, which contains the i-th file's name, download link, and MD5

返回类型

list

static downloadable() → bool

返回

Whether the dataset can be directly downloaded by python codes. If not, the user have to download it manually

返回类型

bool

static extract_downloaded_files(*download_root: str*, *extract_root: str*)

参数

- **download_root** (*str*) –Root directory path which saves downloaded dataset files
- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files

返回

None

This function defines how to extract download files.

static load_origin_data(*file_name: str*) → Dict

参数

file_name (*str*) –path of the events file

返回

a dict whose keys are `['t', 'x', 'y', 'p']` and values are `numpy.ndarray`

返回类型

Dict

This function defines how to read the origin binary data.

static get_H_W() → Tuple

返回

A tuple `(H, W)`, where `H` is the height of the data and `W` is the weight of the data. For example, this function returns `(128, 128)` for the DVS128 Gesture dataset.

返回类型

tuple

static read_aedat_save_to_np(*bin_file: str*, *np_file: str*)

static create_events_np_files(*extract_root: str*, *events_np_root: str*)

参数

- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files
- **events_np_root** –Root directory path which saves events files in the `npz` format

返回

None

This function defines how to convert the origin binary data in `extract_root` to `npz` format and save converted files in `events_np_root`.

spikingjelly.datasets.dvs128_gesture module

class spikingjelly.datasets.dvs128_gesture.**DVS128Gesture**(*root: str*, *train: Optional[bool] = None*, *data_type: str = 'event'*, *frames_number: Optional[int] = None*, *split_by: Optional[str] = None*, *duration: Optional[int] = None*, *custom_integrate_function: Optional[Callable] = None*, *custom_integrated_frames_dir_name: Optional[str] = None*, *transform: Optional[Callable] = None*, *target_transform: Optional[Callable] = None*)

基类：*NeuromorphicDatasetFolder*

The DVS128 Gesture dataset, which is proposed by A Low Power, Fully Event-Based Gesture Recognition System.

Refer to *spikingjelly.datasets.NeuromorphicDatasetFolder* for more details about params information.

static resource_url_md5() → list

返回

A list `url` that `url[i]` is a tuple, which contains the i-th file's name, download link, and MD5

返回类型

list

static downloadable() → bool

返回

Whether the dataset can be directly downloaded by python codes. If not, the user have to download it manually

返回类型

bool

static extract_downloaded_files(*download_root: str*, *extract_root: str*)

参数

- **download_root** (*str*) –Root directory path which saves downloaded dataset files
- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files

返回

None

This function defines how to extract download files.

static load_origin_data(*file_name: str*) → Dict

参数

file_name (*str*) –path of the events file

返回

a dict whose keys are `['t', 'x', 'y', 'p']` and values are `numpy.ndarray`

返回类型

Dict

This function defines how to read the origin binary data.

static split_aedat_files_to_np(*fname: str*, *aedat_file: str*, *csv_file: str*, *output_dir: str*)

static create_events_np_files(*extract_root: str*, *events_np_root: str*)

参数

- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files
- **events_np_root** –Root directory path which saves events files in the `npz` format

返回

None

This function defines how to convert the origin binary data in `extract_root` to `npz` format and save converted files in `events_np_root`.

static get_H_W() → Tuple

返回

A tuple `(H, W)`, where `H` is the height of the data and `W` is the weight of the data. For example, this function returns `(128, 128)` for the DVS128 Gesture dataset.

返回类型

tuple

spikingjelly.datasets.es_imagenet module

spikingjelly.datasets.es_imagenet.**load_events**(*fname: str*)

class spikingjelly.datasets.es_imagenet.**ESImageNet**(*root: str*, *train: Optional[bool] = None*, *data_type: str = 'event'*, *frames_number: Optional[int] = None*, *split_by: Optional[str] = None*, *duration: Optional[int] = None*, *custom_integrate_function: Optional[Callable] = None*, *custom_integrated_frames_dir_name: Optional[str] = None*, *transform: Optional[Callable] = None*, *target_transform: Optional[Callable] = None*)

基类：*NeuromorphicDatasetFolder*

The ES-ImageNet dataset, which is proposed by ES-ImageNet: A Million Event-Stream Classification Dataset for Spiking Neural Networks.

Refer to *spikingjelly.datasets.NeuromorphicDatasetFolder* for more details about params information.

static load_events_np(*fname: str*)

static resource_url_md5() → list

返回

A list `url` that `url[i]` is a tuple, which contains the i-th file's name, download link, and MD5

返回类型

list

static downloadable() → bool

返回

Whether the dataset can be directly downloaded by python codes. If not, the user have to download it manually

返回类型

bool

static extract_downloaded_files(*download_root: str*, *extract_root: str*)

参数

- **download_root** (*str*) –Root directory path which saves downloaded dataset files
- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files

返回

None

This function defines how to extract download files.

static create_events_np_files(*extract_root: str*, *events_np_root: str*)

参数

- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files
- **events_np_root** –Root directory path which saves events files in the `npz` format

返回

None

This function defines how to convert the origin binary data in `extract_root` to `npz` format and save converted files in `events_np_root`.

static get_H_W() → Tuple

返回

A tuple `(H, W)`, where `H` is the height of the data and `W` is the weight of the data. For example, this function returns `(128, 128)` for the DVS128 Gesture dataset.

返回类型

tuple

spikingjelly.datasets.n_caltech101 module

class spikingjelly.datasets.n_caltech101.**NCaltech101**(*root: str*, *data_type: str = 'event'*, *frames_number: Optional[int] = None*, *split_by: Optional[str] = None*, *duration: Optional[int] = None*, *custom_integrate_function: Optional[Callable] = None*, *custom_integrated_frames_dir_name: Optional[str] = None*, *transform: Optional[Callable] = None*, *target_transform: Optional[Callable] = None*)

基类：*NeuromorphicDatasetFolder*

The N-Caltech101 dataset, which is proposed by Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades.

Refer to *spikingjelly.datasets.NeuromorphicDatasetFolder* for more details about params information.

static resource_url_md5() → list

返回

A list `url` that `url[i]` is a tuple, which contains the i-th file's name, download link, and MD5

返回类型

list

static downloadable() → bool

返回

Whether the dataset can be directly downloaded by python codes. If not, the user have to download it manually

返回类型

bool

static extract_downloaded_files(*download_root: str*, *extract_root: str*)

参数

- **download_root** (*str*) –Root directory path which saves downloaded dataset files
- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files

返回

None

This function defines how to extract download files.

static load_origin_data(*file_name: str*) → Dict

参数

file_name (*str*) –path of the events file

返回

a dict whose keys are `['t', 'x', 'y', 'p']` and values are `numpy.ndarray`

返回类型

Dict

This function defines how to read the origin binary data.

static get_H_W() → Tuple

返回

A tuple `(H, W)`, where `H` is the height of the data and `W` is the weight of the

data. For example, this function returns ``(128, 128)`` for the DVS128 Gesture dataset.

返回类型

tuple

static read_bin_save_to_np(*bin_file: str*, *np_file: str*)

static create_events_np_files(*extract_root: str*, *events_np_root: str*)

参数

- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files
- **events_np_root** –Root directory path which saves events files in the `npz` format

返回

None

This function defines how to convert the origin binary data in `extract_root` to `npz` format and save converted files in `events_np_root`.

spikingjelly.datasets.n_mnist module

class spikingjelly.datasets.n_mnist.**NMNIST**(*root: str*, *train: Optional[bool] = None*, *data_type: str = 'event'*, *frames_number: Optional[int] = None*, *split_by: Optional[str] = None*, *duration: Optional[int] = None*, *custom_integrate_function: Optional[Callable] = None*, *custom_integrated_frames_dir_name: Optional[str] = None*, *transform: Optional[Callable] = None*, *target_transform: Optional[Callable] = None*)

基类：*NeuromorphicDatasetFolder*

The N-MNIST dataset, which is proposed by Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades.

Refer to *spikingjelly.datasets.NeuromorphicDatasetFolder* for more details about params information.

static resource_url_md5() → list

返回

A list `url` that `url[i]` is a tuple, which contains the i-th file's name, download link, and MD5

返回类型

list

static downloadable() → bool

返回

Whether the dataset can be directly downloaded by python codes. If not, the user have to download it manually

返回类型

bool

static extract_downloaded_files(*download_root: str*, *extract_root: str*)

参数

- **download_root** (*str*) –Root directory path which saves downloaded dataset files
- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files

返回

None

This function defines how to extract download files.

static load_origin_data(*file_name: str*) → Dict

参数

file_name (*str*) –path of the events file

返回

a dict whose keys are `['t', 'x', 'y', 'p']` and values are `numpy.ndarray`

返回类型

Dict

This function defines how to read the origin binary data.

static get_H_W() → Tuple

返回

A tuple `(H, W)`, where `H` is the height of the data and `W` is the weight of the data. For example, this function returns `(128, 128)` for the DVS128 Gesture dataset.

返回类型

tuple

static read_bin_save_to_np(*bin_file: str*, *np_file: str*)

static create_events_np_files(*extract_root: str*, *events_np_root: str*)

参数

- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files
- **events_np_root** –Root directory path which saves events files in the `npz` format

返回

None

This function defines how to convert the origin binary data in `extract_root` to `npz` format and save converted files in `events_np_root`.

spikingjelly.datasets.nav_gesture module

spikingjelly.datasets.nav_gesture.**peek**(*f*, *length=1*)

spikingjelly.datasets.nav_gesture.**readATIS_tddat**(*file_name*, *orig_at_zero=True*, *drop_negative_dt=True*, *verbose=True*, *events_restriction=[0, inf]*)

reads ATIS td events in .dat format

input: filename: string, path to the .dat file orig_at_zero: bool, if True, timestamps will start at 0 drop_negative_dt: bool, if True, events with a timestamp greater than the previous event are dismissed verbose: bool, if True, verbose mode. events_restriction: list [min ts, max ts], will return only events with ts in the defined boundaries

output: timestamps: numpy array of length (number of events), timestamps coords: numpy array of size (number of events, 2), spatial coordinates: col 0 is x, col 1 is y. polarities: numpy array of length (number of events), polarities removed_events: integer, number of removed events (negative delta-ts)

class spikingjelly.datasets.nav_gesture.**NAVGestureWalk**(*root: str*, *data_type: str = 'event'*, *frames_number: Optional[int] = None*, *split_by: Optional[str] = None*, *duration: Optional[int] = None*, *custom_integrate_function: Optional[Callable] = None*, *custom_integrated_frames_dir_name: Optional[str] = None*, *transform: Optional[Callable] = None*, *target_transform: Optional[Callable] = None*)

基类：*NeuromorphicDatasetFolder*

The Nav Gesture dataset, which is proposed by Event-Based Gesture Recognition With Dynamic Background Suppression Using Smartphone Computational Capabilities.

Refer to *spikingjelly.datasets.NeuromorphicDatasetFolder* for more details about params information.

static resource_url_md5() → list

返回

A list `url` that `url[i]` is a tuple, which contains the i-th file's name, download link, and MD5

返回类型

list

static downloadable() → bool

返回

Whether the dataset can be directly downloaded by python codes. If not, the user have to download it manually

返回类型

bool

static extract_downloaded_files(*download_root: str*, *extract_root: str*)

参数

- **download_root** (*str*) –Root directory path which saves downloaded dataset files
- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files

返回

None

This function defines how to extract download files.

static get_H_W() → Tuple

返回

A tuple `(H, W)`, where `H` is the height of the data and `W` is the weight of the data. For example, this function returns ``(128, 128)` for the DVS128 Gesture dataset.

返回类型

tuple

static read_aedat_save_to_np(*bin_file: str*, *np_file: str*)

static create_events_np_files(*extract_root: str*, *events_np_root: str*)

参数

- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files
- **events_np_root** –Root directory path which saves events files in the `npz` format

返回

None

This function defines how to convert the origin binary data in `extract_root` to `npz` format and save converted files in `events_np_root`.

class spikingjelly.datasets.nav_gesture.**NAVGestureSit**(*root: str*, *data_type: str = 'event'*, *frames_number: Optional[int] = None*, *split_by: Optional[str] = None*, *duration: Optional[int] = None*, *custom_integrate_function: Optional[Callable] = None*, *custom_integrated_frames_dir_name: Optional[str] = None*, *transform: Optional[Callable] = None*, *target_transform: Optional[Callable] = None*)

基类：*NAVGestureWalk*

The Nav Gesture dataset, which is proposed by Event-Based Gesture Recognition With Dynamic Background Suppression Using Smartphone Computational Capabilities.

Refer to *spikingjelly.datasets.NeuromorphicDatasetFolder* for more details about params information.

static resource_url_md5() → list

返回

A list `url` that `url[i]` is a tuple, which contains the i-th file's name, download link, and MD5

返回类型

list

static extract_downloaded_files(*download_root: str*, *extract_root: str*)

参数

- **download_root** (*str*) –Root directory path which saves downloaded dataset files
- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files

返回

None

This function defines how to extract download files.

spikingjelly.datasets.speechcommands module

spikingjelly.datasets.speechcommands.**load_speechcommands_item**(*relpath: str*, *path: str*) → Tuple[Tensor, int, str, str, int]

class spikingjelly.datasets.speechcommands.**SPEECHCOMMANDS**(*label_dict: Dict*, *root: str*, *silence_cnt: Optional[int] = 0*, *silence_size: Optional[int] = 16000*, *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：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
- **silence_cnt** (*int, optional*) –Silence 数据的数量
- **silence_size** (*int, optional*) –Silence 数据的尺寸
- **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. 辅助词，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。

Module contents

spikingjelly.datasets.**play_frame**(*x: Tensor*, *save_gif_to: Optional[str] = None*) → None

参数

- **x** (*torch.Tensor or np.ndarray*) –frames with `shape=[T, 2, H, W]`
- **save_gif_to** (*str*) –If `None`, this function will play the frames. If `True`, this function will not play the frames but save frames to a gif file in the directory `save_gif_to`

返回

None

spikingjelly.datasets.**load_matlab_mat**(*file_name: str*) → Dict

参数

file_name (*str*) –path of the matlab’s mat file

返回

a dict whose keys are `['t', 'x', 'y', 'p']` and values are `numpy.ndarray`

返回类型

Dict

spikingjelly.datasets.**load_aedat_v3**(*file_name: str*) → Dict

参数

file_name (*str*) –path of the aedat v3 file

返回

a dict whose keys are `['t', 'x', 'y', 'p']` and values are `numpy.ndarray`

返回类型

Dict

This function is written by referring to https://gitlab.com/inivation/dv/dv-python . It can be used for DVS128 Gesture.

spikingjelly.datasets.**load_ATIS_bin**(*file_name: str*) → Dict

参数

file_name (*str*) –path of the aedat v3 file

返回

a dict whose keys are `['t', 'x', 'y', 'p']` and values are `numpy.ndarray`

返回类型

Dict

This function is written by referring to https://github.com/jackd/events-tfds . Each ATIS binary 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)

spikingjelly.datasets.**load_npz_frames**(*file_name: str*) → ndarray

参数

file_name (*str*) –path of the npz file that saves the frames

返回

frames

返回类型

np.ndarray

spikingjelly.datasets.**integrate_events_segment_to_frame**(*x: ndarray*, *y: ndarray*, *p: ndarray*, *H: int*, *W: int*, *j_l: int = 0*, *j_r: int = -1*) → ndarray

param x

x-coordinate of events

type x

numpy.ndarray

param y

y-coordinate of events

type y

numpy.ndarray

param p

polarity of events

type p

numpy.ndarray

param H

height of the frame

type H
int

param W
weight of the frame

type W
int

param j_l
the start index of the integral interval, which is included

type j_l
int

param j_r
the right index of the integral interval, which is not included

type j_r
return
frames

rtype
np.ndarray

Denote a two channels frame as F and a pixel at (p, x, y) as $F(p, x, y)$, the pixel value is integrated from the events data whose indices are in $[j_l, j_r)$: .. math:

```
F(p, x, y) = \sum_{i = j_{l}}^{j_{r} - 1} \mathcal{I}_{p, x, y}(p_{i}, x_
↪{i}, y_{i})
```

where :math:‘lfloor cdot

floor‘ 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.datasets.**cal_fixed_frames_number_segment_index**(*events_t: ndarray*, *split_by: str*, *frames_num: int*) → tuple

参数

- **events_t** (*numpy.ndarray*) –events’ t
- **split_by** (*str*) – ‘time’ or ‘number’
- **frames_num** (*int*) –the number of frames

返回
a tuple `(j_l, j_r)`

返回类型

tuple

Denote `frames_num` as M, if `split_by` is `'time'`, then .. math:

```
\Delta T & = [\frac{t_{N-1} - t_{0}}{M}] \\
j_{l} & = \mathop{\arg\min}\limits_{k} \{t_{k} | t_{k} \geq t_{0} + \Delta T \
↪cdot j\} \\
j_{r} & = \begin{cases} \mathop{\arg\max}\limits_{k} \{t_{k} | t_{k} < t_{0} + \
↪Delta T \cdot (j + 1)\} + 1, & j <  M - 1 \cr N, & j = M - 1 \end{cases}
```

If `split_by` is `'number'`, then .. math:

```
j_{l} & = [\frac{N}{M}] \cdot j \\
j_{r} & = \begin{cases} [\frac{N}{M}] \cdot (j + 1), & j <  M - 1 \cr N, & j = M -
↪ 1 \end{cases}
```

spikingjelly.datasets.**integrate_events_by_fixed_frames_number**(*events: Dict*, *split_by: str*, *frames_num: int*, *H: int*, *W: int*) → ndarray

参数

- **events** (*Dict*) –a dict whose keys are `['t', 'x', 'y', 'p']` and values are `numpy.ndarray`
- **split_by** (*str*) – 'time' or 'number'
- **frames_num** (*int*) –the number of frames
- **H** (*int*) –the height of frame
- **W** (*int*) –the weight of frame

返回

frames

返回类型

np.ndarray

Integrate events to frames by fixed frames number. See *cal_fixed_frames_number_segment_index* and *integrate_events_segment_to_frame* for more details.

spikingjelly.datasets.**integrate_events_file_to_frames_file_by_fixed_frames_number**(*loader: Callable*, *events_np_fi str*, *out- put_dir: str*, *split_by: str*, *frames_num int*, *H: int*, *W: int*, *print_save: bool = False*) → None

参数

- **loader** (*Callable*) –a function that can load events from *events_np_file*
- **events_np_file** (*str*) –path of the events np file
- **output_dir** (*str*) –output directory for saving the frames
- **split_by** (*str*) – 'time' or 'number'
- **frames_num** (*int*) –the number of frames
- **H** (*int*) –the height of frame
- **W** (*int*) –the weight of frame
- **print_save** (*bool*) –If `True`, this function will print saved files' paths.

返回

None

Integrate a events file to frames by fixed frames number and save it. See *cal_fixed_frames_number_segment_index* and *integrate_events_segment_to_frame* for more details.

spikingjelly.datasets.**integrate_events_by_fixed_duration**(*events: Dict*, *duration: int*, *H: int*, *W: int*) → ndarray

参数

- **events** (*Dict*) –a dict whose keys are `['t', 'x', 'y', 'p']` and values are `numpy.ndarray`
- **duration** (*int*) –the time duration of each frame
- **H** (*int*) –the height of frame
- **W** (*int*) –the weight of frame

返回

frames

返回类型

np.ndarray

Integrate events to frames by fixed time duration of each frame.

spikingjelly.datasets.**integrate_events_file_to_frames_file_by_fixed_duration**(*loader: Callable*, *events_np_file: str*, *output_dir: str*, *duration: int*, *H: int*, *W: int*, *print_save: bool = False*) → None

参数

- **loader** (*Callable*) –a function that can load events from *events_np_file*
- **events_np_file** (*str*) –path of the events np file

- **output_dir** (*str*) –output directory for saving the frames
- **duration** (*int*) –the time duration of each frame
- **H** (*int*) –the height of frame
- **W** (*int*) –the weight of frame
- **print_save** (*bool*) –If `True`, this function will print saved files' paths.

返回

None

Integrate events to frames by fixed time duration of each frame.

spikingjelly.datasets.**save_frames_to_npz_and_print**(*fname: str*, *frames*)

spikingjelly.datasets.**create_same_directory_structure**(*source_dir: str*, *target_dir: str*) → None

参数

- **source_dir** (*str*) –Path of the directory that be copied from
- **target_dir** (*str*) –Path of the directory that be copied to

返回

None

Create the same directory structure in `target_dir` with that of `source_dir`.

spikingjelly.datasets.**split_to_train_test_set**(*train_ratio: float*, *origin_dataset: Dataset*, *num_classes: int*, *random_split: bool = False*)

参数

- **train_ratio** (*float*) –split the ratio of the origin dataset as the train set
- **origin_dataset** (*torch.utils.data.Dataset*) –the origin dataset
- **num_classes** (*int*) –total classes number, e.g., `10` for the MNIST dataset
- **random_split** (*int*) –If `False`, the front ratio of samples in each classes will be included in train set, while the reset will be included in test set. If `True`, this function will split samples in each classes randomly. The randomness is controlled by `numpy.random.seed`

返回

a tuple `(train_set, test_set)`

返回类型

tuple

spikingjelly.datasets.**pad_sequence_collate**(*batch: list*)

参数

batch (*list*) –a list of samples that contains `(x, y)`, where `x` is a list containing sequences with different length and `y` is the label

返回

batched samples `(x_p, y, x_len)`, where ``x_p` is padded `x` with the same length, *y*‘ is the label, and `x_len` is the length of the `x`

返回类型

tuple

This function can be use as the `collate_fn` for `DataLoader` to process the dataset with variable length, e.g., a `NeuromorphicDatasetFolder` with fixed duration to integrate events to frames. Here is an example: .. code-block:: python class VariableLengthDataset(torch.utils.data.Dataset):

def __init__(self, n=1000):

super().__init__() self.n = n

def __getitem__(self, i):

return torch.rand([i + 1, 2]), self.n - i - 1

def __len__(self):

return self.n

loader = torch.utils.data.DataLoader(VariableLengthDataset(n=32), batch_size=2,

collate_fn=pad_sequence_collate,
shuffle=True)

for i, (x_p, label, x_len) in enumerate(loader):

print(f’ x_p.shape={x_p.shape}, label={label}, x_len={x_len}’) if i == 2:

break

And the outputs are: .. code-block:: bash

x_p.shape=torch.Size([2, 18, 2]), label=tensor([14, 30]), x_len=tensor([18, 2]) x_p.shape=torch.Size([2, 29, 2]), label=tensor([3, 6]), x_len=tensor([29, 26]) x_p.shape=torch.Size([2, 23, 2]), label=tensor([9, 23]), x_len=tensor([23, 9])

`spikingjelly.datasets.`**padded_sequence_mask**(*sequence_len: Tensor*, *T=None*)

参数

- **sequence_len** (*torch.Tensor*) –a tensor `shape = [N]` that contains sequences lengths of each batch element
- **T** (*int*) –The maximum length of sequences. If `None`, the maximum element in `sequence_len` will be seen as `T`

返回

a bool mask with shape = [T, N], where the padded position is `False`

返回类型

torch.Tensor

Here is an example: .. code-block:: python

x1 = torch.rand([2, 6]) x2 = torch.rand([3, 6]) x3 = torch.rand([4, 6]) x = torch.nn.utils.rnn.pad_sequence([x1, x2, x3]) # [T, N, *] print('x.shape=', x.shape) x_len = torch.as_tensor([x1.shape[0], x2.shape[0], x3.shape[0]]) mask = padded_sequence_mask(x_len) print('mask.shape=', mask.shape) print('mask=n', mask)

And the outputs are: .. code-block:: bash

x.shape= torch.Size([4, 3, 6]) mask.shape= torch.Size([4, 3]) mask=

tensor([[True, True, True],
[True, True, True], [False, True, True], [False, False, True]])

class spikingjelly.datasets.**NeuromorphicDatasetFolder**(*root: str*, *train: Optional[bool] = None*, *data_type: str = 'event'*, *frames_number: Optional[int] = None*, *split_by: Optional[str] = None*, *duration: Optional[int] = None*, *custom_integrate_function: Optional[Callable] = None*, *custom_integrated_frames_dir_name: Optional[str] = None*, *transform: Optional[Callable] = None*, *target_transform: Optional[Callable] = None*)

基类：DatasetFolder

参数

- **root** (*str*) –root path of the dataset
- **train** (*bool*) –whether use the train set. Set `True` or `False` for those datasets provide train/test division, e.g., DVS128 Gesture dataset. If the dataset does not provide train/test division, e.g., CIFAR10-DVS, please set `None` and use *split_to_train_test_set* function to get train/test set
- **data_type** (*str*) –*event* or *frame*
- **frames_number** (*int*) –the integrated frame number
- **split_by** (*str*) –*time* or *number*
- **duration** (*int*) –the time duration of each frame
- **custom_integrate_function** (*Callable*) –a user-defined function that inputs are `events, H, W`. `events` is a dict whose keys are `['t', 'x', 'y', 'p']` and values

are `numpy.ndarray` `H` is the height of the data and `W` is the weight of the data. For example, H=128 and W=128 for the DVS128 Gesture dataset. The user should define how to integrate events to frames, and return frames.

- **custom_integrated_frames_dir_name** (*str or None*) –The name of directory for saving frames integrating by `custom_integrate_function`. If `custom_integrated_frames_dir_name` is `None`, it will be set to `custom_integrate_function.__name__`
- **transform** (*callable*) –a function/transform that takes in a sample and returns a transformed version. E.g, `transforms.RandomCrop` for images.
- **target_transform** (*callable*) –a function/transform that takes in the target and transforms it.

The base class for neuromorphic dataset. Users can define a new dataset by inheriting this class and implementing all abstract methods. Users can refer to *spikingjelly.datasets.dvs128_gesture.DVS128Gesture*. If `data_type == 'event'`

the sample in this dataset is a dict whose keys are `['t', 'x', 'y', 'p']` and values are `numpy.ndarray`.

If `data_type == 'frame'` and `frames_number` is not `None`

events will be integrated to frames with fixed frames number. `split_by` will define how to split events. See *cal_fixed_frames_number_segment_index* for more details.

If `data_type == 'frame'`, `frames_number` is `None`, and `duration` is not `None`

events will be integrated to frames with fixed time duration.

If `data_type == 'frame'`, `frames_number` is `None`, `duration` is `None`, and `custom_integrate_function` is not `None`:

events will be integrated by the user-defined function and saved to the `custom_integrated_frames_dir_name` directory in `root` directory. Here is an example from SpikingJelly’s tutorials: .. code-block:: python

from spikingjelly.datasets.dvs128_gesture import DVS128Gesture from typing import Dict import numpy as np import spikingjelly.datasets as sjds def integrate_events_to_2_frames_randomly(events: Dict, H: int, W: int):

index_split = np.random.randint(low=0, high=events[‘t’].__len__()) frames = np.zeros([2, 2, H, W]) t, x, y, p = (events[key] for key in (‘t’, ‘x’, ‘y’, ‘p’)) frames[0] = sjds.integrate_events_segment_to_frame(x, y, p, H, W, 0, index_split) frames[1] = sjds.integrate_events_segment_to_frame(x, y, p, H, W, index_split, events[‘t’].__len__()) return frames

root_dir = ‘D:/datasets/DVS128Gesture’ train_set = DVS128Gesture(root_dir, train=True, data_type=’frame’, custom_integrate_function=integrate_events_to_2_frames_randomly) from spikingjelly.datasets import play_frame frame, label = train_set[500] play_frame(frame)

abstract static resource_url_md5() → list

返回

A list `url` that `url[i]` is a tuple, which contains the i-th file's name, download link, and MD5

返回类型

list

abstract static downloadable() → bool

返回

Whether the dataset can be directly downloaded by python codes. If not, the user have to download it manually

返回类型

bool

abstract static extract_downloaded_files(*download_root: str*, *extract_root: str*)

参数

- **download_root** (*str*) –Root directory path which saves downloaded dataset files
- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files

返回

None

This function defines how to extract download files.

abstract static create_events_np_files(*extract_root: str*, *events_np_root: str*)

参数

- **extract_root** (*str*) –Root directory path which saves extracted files from downloaded files
- **events_np_root** –Root directory path which saves events files in the `npz` format

返回

None

This function defines how to convert the origin binary data in `extract_root` to `npz` format and save converted files in `events_np_root`.

abstract static get_H_W() → Tuple

返回

A tuple `(H, W)`, where `H` is the height of the data and `W` is the weight of the data. For example, this function returns `(128, 128)` for the DVS128 Gesture dataset.

返回类型

tuple

static load_events_np(*fname: str*)

参数

fname –file name

返回

a dict whose keys are `['t', 'x', 'y', 'p']` and values are `numpy.ndarray`

This function defines how to load a sample from *events_np*. In most cases, this function is *np.load*. But for some datasets, e.g., ES-ImageNet, it can be different.

spikingjelly.datasets.**random_temporal_delete**(*x_seq: Tensor*, *T_remain: int*, *batch_first*)

参数

- **x_seq** (*torch.Tensor or np.ndarray*) –a sequence with *shape = [T, N, *]*, where *T* is the sequence length and *N* is the batch size
- **T_remain** (*int*) –the remained length
- **batch_first** (*bool*) –if *True*, *x_seq* will be regarded as *shape = [N, T, *]*

返回

the sequence with length *T_remain*, which is obtained by randomly removing *T - T_remain* slices

返回类型

torch.Tensor or np.ndarray

The random temporal delete data augmentation used in Deep Residual Learning in Spiking Neural Networks. Codes example: .. code-block:: python

import torch from spikingjelly.datasets import random_temporal_delete T = 8 T_remain = 5 N = 4 x_seq = torch.arange(0, N*T).view([N, T]) print(‘x_seq=n’ , x_seq) print(‘random_temporal_delete(x_seq)=n’ , random_temporal_delete(x_seq, T_remain, batch_first=True))

Outputs: .. code-block:: shell

x_seq=

tensor([[0, 1, 2, 3, 4, 5, 6, 7],
[8, 9, 10, 11, 12, 13, 14, 15], [16, 17, 18, 19, 20, 21, 22, 23], [24, 25, 26, 27, 28, 29, 30, 31]])

random_temporal_delete(x_seq)=

tensor([[0, 1, 4, 6, 7],
[8, 9, 12, 14, 15], [16, 17, 20, 22, 23], [24, 25, 28, 30, 31]])

class spikingjelly.datasets.**RandomTemporalDelete**(*T_remain: int*, *batch_first: bool*)

基类：Module

参数

- **T_remain** (*int*) –the remained length
- **batch_first** –if *True*, *x_seq* will be regarded as *shape = [N, T, *]*

The random temporal delete data augmentation used in Deep Residual Learning in Spiking Neural Networks. Refer to *random_temporal_delete* for more details.

forward(*x_seq: Tensor*)

training: bool

spikingjelly.datasets.**create_sub_dataset**(*source_dir: str*, *target_dir: str*, *ratio: float*, *use_soft_link=True*, *randomly=False*)

参数

- **source_dir** (*str*) –the directory path of the origin dataset
- **target_dir** (*str*) –the directory path of the sub dataset
- **ratio** (*float*) –the ratio of samples sub dataset will copy from the origin dataset
- **use_soft_link** (*bool*) –if `True`, the sub dataset will use soft link to copy; else, the sub dataset will copy files
- **randomly** (*bool*) –if `True`, the files copy from the origin dataset will be picked up randomly. The randomness is controlled by `numpy.random.seed`

Create a sub dataset with copy `ratio` of samples from the origin dataset.

7.5.3 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：Module

forward(*x: Tensor*)

training: bool

spikingjelly.event_driven.examples.tempotron_mnist.**main**()

返回

None

- *API in English*

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。

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*

Use Gaussian tuned activation function encoder to encode the images to spikes.

The network with single Tempotron structure for classifying MNIST.

This function initials the network, starts training and shows accuracy on test dataset.

Module contents

spikingjelly.event_driven.encoding package

Module contents

class spikingjelly.event_driven.encoding.**GaussianTuning**(*n*, *m*, *x_min: Tensor*, *x_max: 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: 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：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: 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: 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

7.5.4 spikingjelly.visualizing package

Module contents

spikingjelly.visualizing.**plot_2d_heatmap**(*array: 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=[T, N] 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

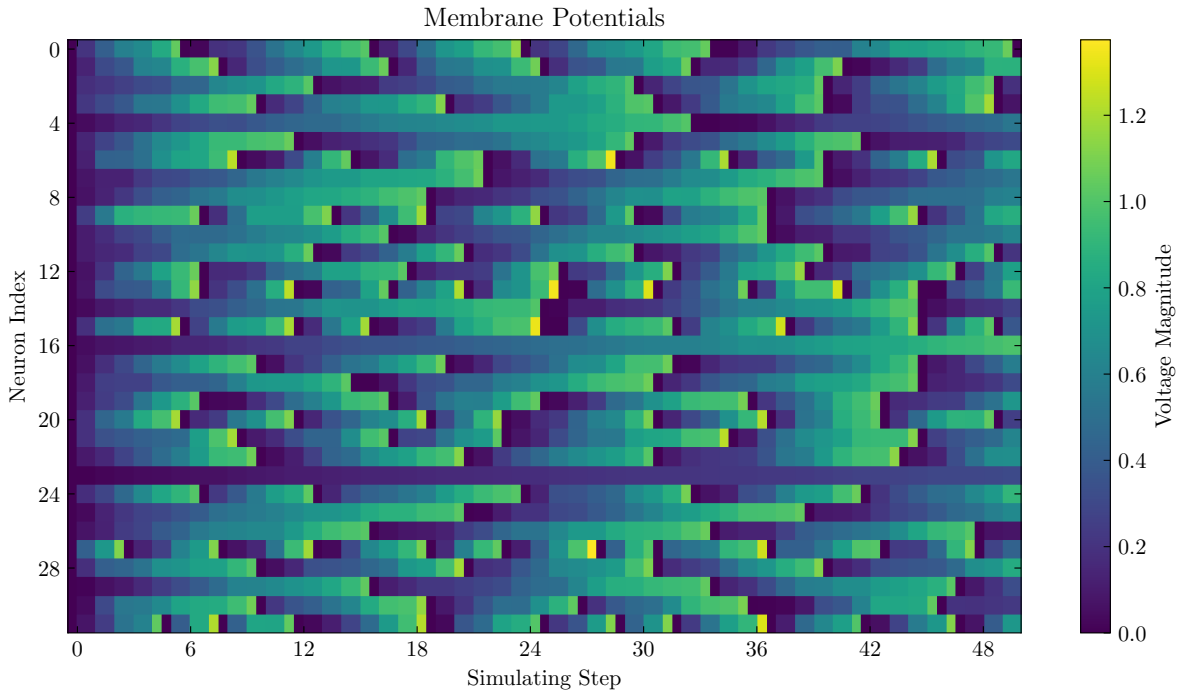
- **x_max** –横轴的最大刻度。若设置为 None，则认为横轴的最大刻度是 array.shape[1]
- **dpi** –绘图的 dpi

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：

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

lif = neuron.LIFNode(tau=100.)
x = torch.rand(size=[32]) * 4
T = 50
s_list = []
v_list = []
for t in range(T):
    s_list.append(lif(x).unsqueeze(0))
    v_list.append(lif.v.unsqueeze(0))

s_list = torch.cat(s_list)
v_list = torch.cat(v_list)

visualizing.plot_2d_heatmap(array=np.asarray(v_list), title='Membrane Potentials',
↪ xlabel='Simulating Step',
                            ylabel='Neuron Index', int_x_ticks=True, x_max=T,␣
↪dpi=200)
plt.show()
```



spikingjelly.visualizing.**plot_2d_bar_in_3d**(*array: 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

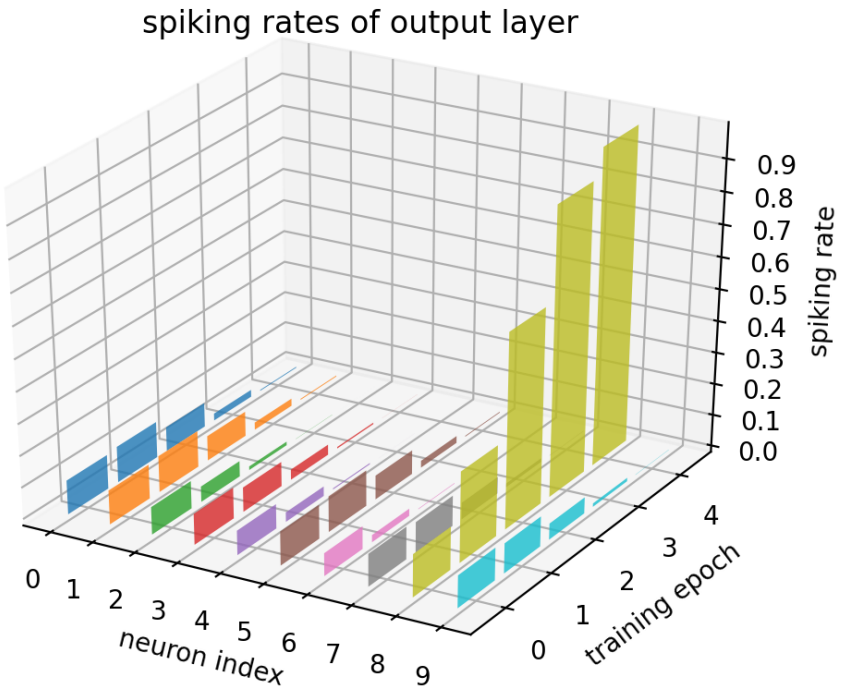
- **array** –shape=[T, N] 的任意数组
- **title** –图的标题
- **xlabel** –x 轴的 label
- **ylabel** –y 轴的 label
- **zlabel** –z 轴的 label
- **int_x_ticks** –x 轴上是否只显示整数刻度
- **int_y_ticks** –y 轴上是否只显示整数刻度
- **int_z_ticks** –z 轴上是否只显示整数刻度
- **dpi** –绘图的 dpi

返回

绘制好的 figure

将 shape=[T, N] 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：

```
import torch
from spikingjelly import visualizing
from matplotlib import pyplot as plt

Epochs = 5
N = 10
firing_rate = torch.zeros(Epochs, N)
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()
```



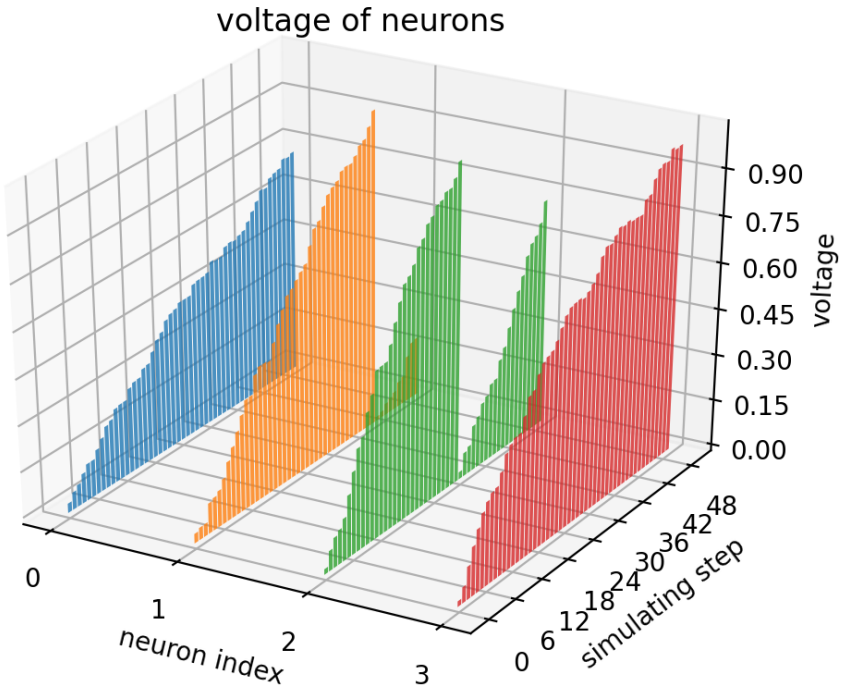
也可以用来绘制一张表示多个神经元在不同时刻的电压的热力图，示例代码：

```
import torch
```

(续下页)

(接上页)

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

neuron_num = 4
T = 50
lif_node = neuron.LIFNode(tau=100.)
w = torch.rand([neuron_num]) * 10
v_list = []
for t in range(T):
    lif_node(w * torch.rand(size=[neuron_num]))
    v_list.append(lif_node.v.unsqueeze(0))

v_list = torch.cat(v_list)
visualizing.plot_2d_bar_in_3d(v_list, 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: 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=[T, N] 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

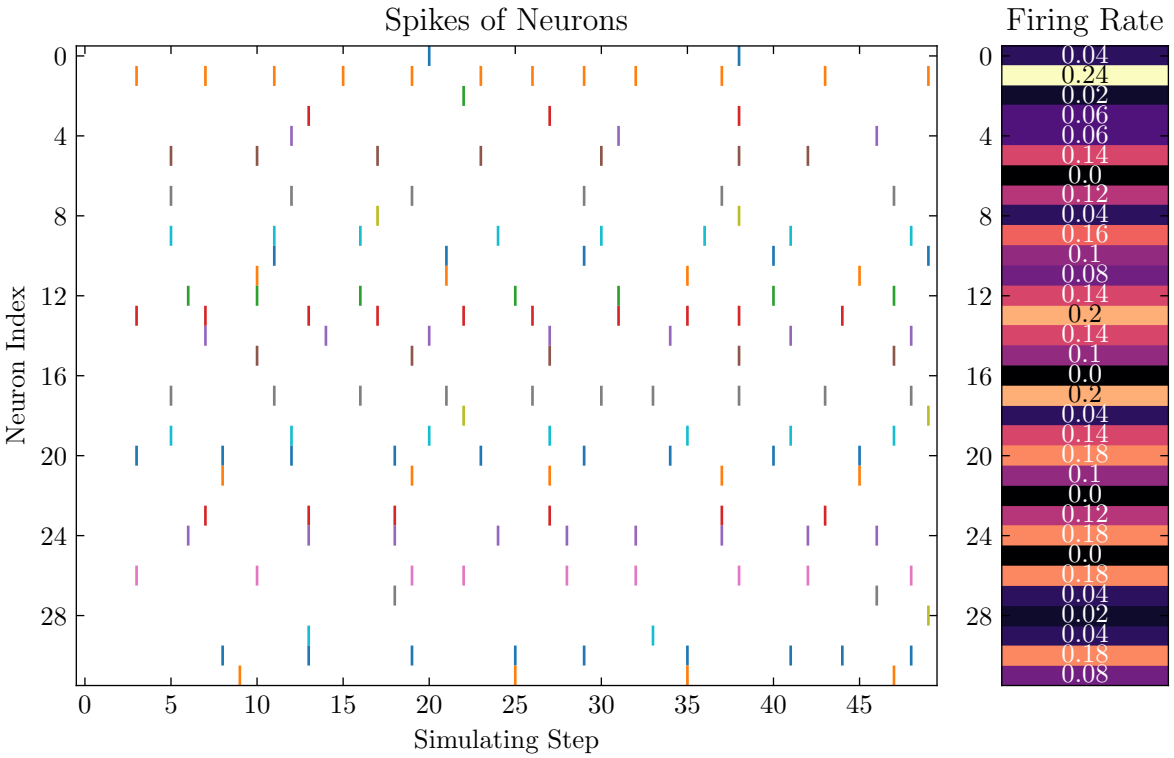
画出 N 个时长为 T 的脉冲数据。可以用来画 N 个神经元在 T 个时刻的脉冲发放情况，示例代码：

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

lif = neuron.LIFNode(tau=100.)
x = torch.rand(size=[32]) * 4
T = 50
s_list = []
v_list = []
for t in range(T):
    s_list.append(lif(x).unsqueeze(0))
    v_list.append(lif.v.unsqueeze(0))

s_list = torch.cat(s_list)
v_list = torch.cat(v_list)

visualizing.plot_1d_spikes(spikes=np.asarray(s_list), title='Membrane Potentials',
↪ xlabel='Simulating Step',
```

(续下页)

(接上页)

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



spikingjelly.visualizing.**plot_2d_spiking_feature_map**(*spikes: asarray*, *nrows*, *ncols*, *space*, *title: str*, *dpi=200*)

参数

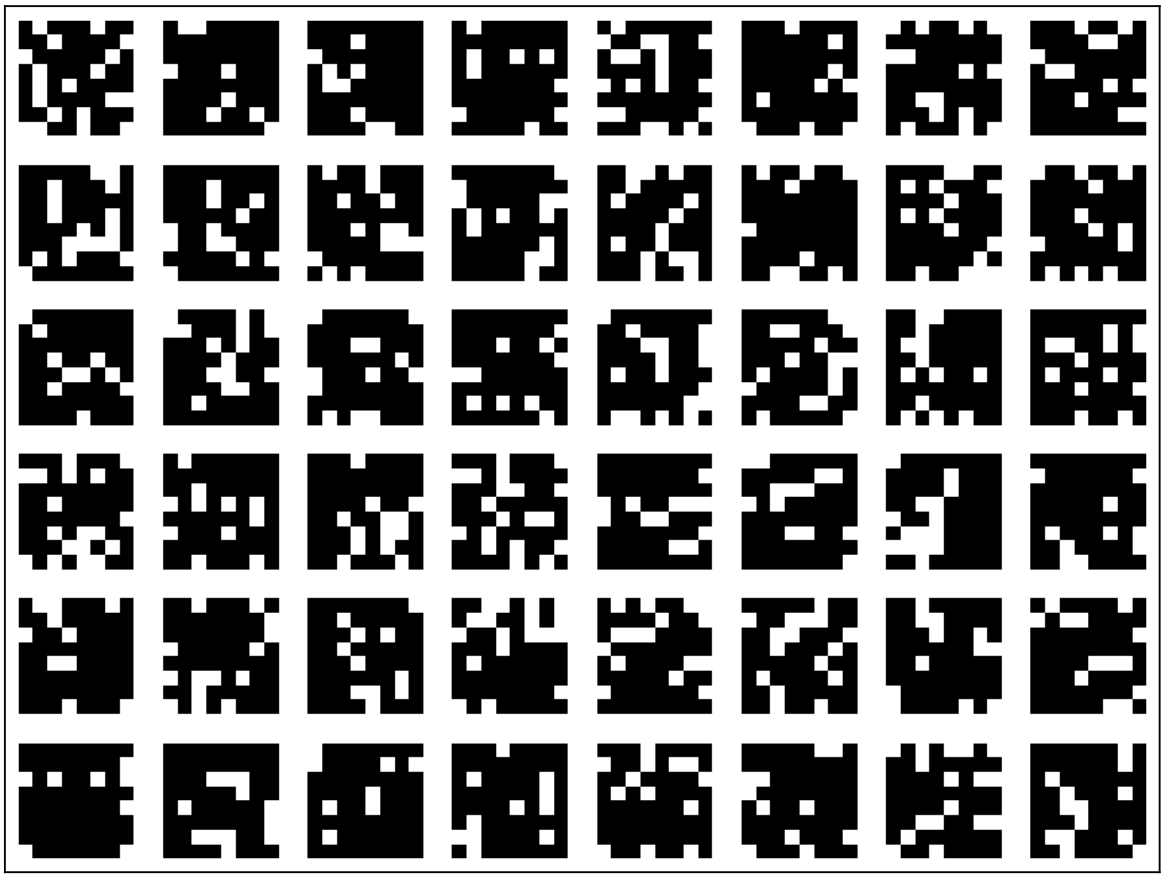
- **spikes** –shape=[C, W, H]，C 个尺寸为 W * H 的脉冲矩阵，矩阵中的元素为 0 或 1。这样的矩阵一般来源于卷积层后的脉冲神经元的输出
- **nrows** –画成多少行
- **ncols** –画成多少列
- **space** –矩阵之间的间隙
- **title** –图的标题
- **dpi** –绘图的 dpi

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
from matplotlib import pyplot as plt

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: ndarray*, *s: ndarray*, *v_threshold=1.0*, *v_reset=0.0*, *title='V_{t} and S_{t} of the neuron'*, *dpi=200*)

参数

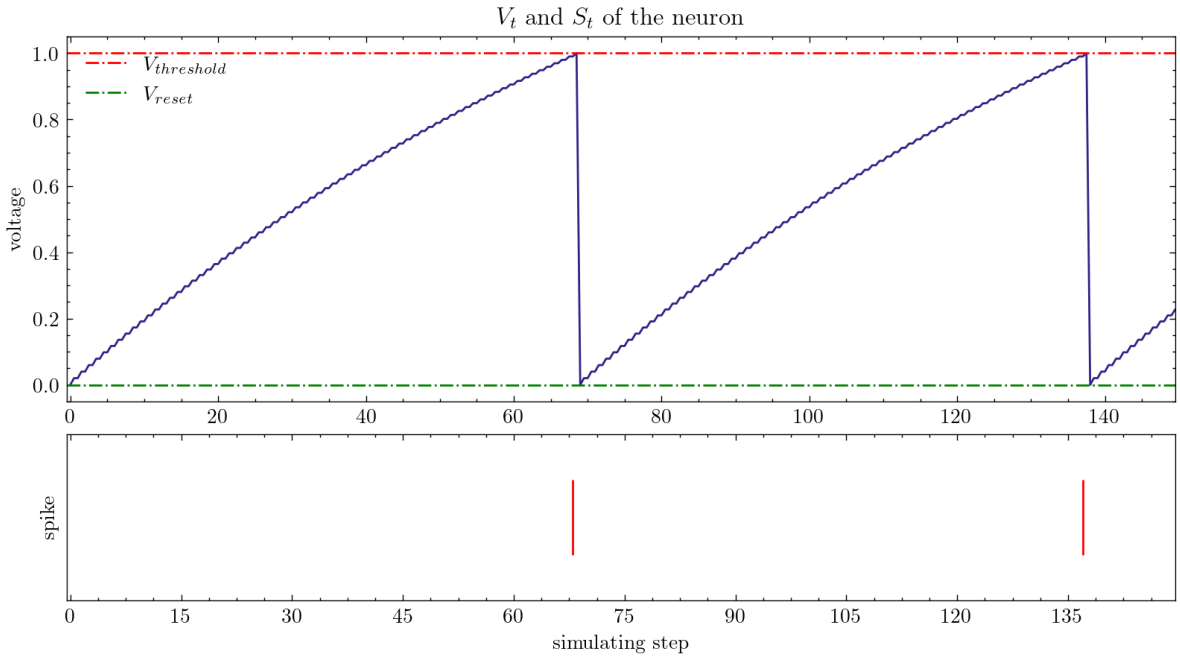
- **v** –shape=[T], 存放神经元不同时刻的电压
- **s** –shape=[T], 存放神经元不同时刻释放的脉冲
- **v_threshold** –神经元的阈值电压
- **v_reset** –神经元的重置电压。也可以为 None
- **title** –图的标题
- **dpi** –绘图的 dpi

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。示例代码：

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

lif = neuron.LIFNode(tau=100.)
x = torch.Tensor([2.0])
T = 150
s_list = []
v_list = []
for t in range(T):
    s_list.append(lif(x))
    v_list.append(lif.v)
visualizing.plot_one_neuron_v_s(v_list, s_list, v_threshold=lif.v_threshold, v_
↪reset=lif.v_reset,
                                dpi=200)
plt.show()
```



7.5.5 spikingjelly.cext package

spikingjelly.cext.functional package

Module contents

class spikingjelly.cext.functional.**sparse_mm_dense_atf**

基类：Function

static forward(*ctx*, *sparse: Tensor*, *dense: Tensor*)

static backward(*ctx*, *grad_output*)

spikingjelly.cext.functional.**sparse_mm_dense**(*sparse: Tensor*, *dense: 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：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: Tensor*) → Tensor

in_features: int

out_features: int

weight: Tensor

class spikingjelly.cext.layer.**AutoSparseLinear**(*in_features: int*, *out_features: int*, *bias: bool = True*, *in_spikes: bool = False*)

基类：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: Tensor*) → 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: Tensor

spikingjelly.cext.neuron package

Module contents

Module contents

spikingjelly.cext.**cal_fun_t**(*n*, *device*, *f*, **args*, ***kwargs*)

Python 模块索引

S

索引

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

V

W

模块