SpikingFlow

发布 *0.2.2*

PKU MLG

2020 年 11 月 21 日

Contents

SpikingFlow 是一个基于 PyTorch 的脉冲神经网络 (Spiking Neuron Network, SNN) 框架。

CHAPTER 1

安装

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

推荐从 GitHub 下载源代码，因为 pip 上的版本通常更新慢，bug 可能比较多。

从 pip 安装：

```
pip install SpikingFlow
```

或者对于开发者，下载源代码，进行代码补充、修改和测试：

```
git clone https://github.com/fangwei123456/SpikingFlow.git
```

CHAPTER 2

快速上手教程

`SpikingFlow.softbp` 使用时间驱动仿真 SNN，使用反向传播、梯度下降来学习。

`SpikingFlow.event_driven` 是使用事件驱动仿真 SNN，使用反向传播、梯度下降来学习。

而其他的 `SpikingFlow.*` 是使用事件驱动仿真 SNN，使用生物可解释性的方法（例如 STDP）来学习。

因此，`SpikingFlow.softbp` 和 `SpikingFlow.event_driven` 以及其他的 `SpikingFlow.*` 包，三者是平行关系，互不交叉。例如使用者对 `SpikingFlow.softbp` 感兴趣，他只需要阅读 `SpikingFlow.softbp` 相关的教程或源代码就可以上手。

- *神经元 SpikingFlow.neuron*
- *编码器 SpikingFlow.encoding*
- *仿真器 SpikingFlow.simulating*
- *突触连接 SpikingFlow.connection*
- *学习规则 SpikingFlow.learning*
- *软反向传播 SpikingFlow.softbp*
- *事件驱动 SpikingFlow.event_driven*

CHAPTER 3

模块文档

3.1 SpikingFlow

3.1.1 SpikingFlow package

3.1.1.1 Subpackages

3.1.1.1.1 SpikingFlow.connection package

Submodules

SpikingFlow.connection.transform module

class SpikingFlow.connection.transform.**BaseTransformer**

基类：torch.nn.modules.module.Module

脉冲-电流转换器的基类

输入是脉冲（torch.bool），输出是电流（torch.float）

forward(*in_spike*)

参数 **in_spike** –输入脉冲

返回 输出电流

要求子类必须实现这个函数

reset()

返回 None

重置所有的状态变量为初始值对于有状态的子类，必须实现这个函数

training: bool

class SpikingFlow.connection.transform.**SpikeCurrent**(*amplitude=1*)

基类：*SpikingFlow.connection.transform.BaseTransformer*

参数 **amplitude** –放大系数

输入脉冲，输出与脉冲形状完全相同、离散的、大小为 amplitude 倍的电流

forward(*in_spike*)

参数 **in_spike** –输入脉冲

返回 输出电流

简单地将输入脉冲转换为 0/1 的浮点数，然后乘以 amplitude

training: bool

class SpikingFlow.connection.transform.**ExpDecayCurrent**(*tau*, *amplitude=1*)

基类：*SpikingFlow.connection.transform.BaseTransformer*

参数

- **tau** –衰减的时间常数，越小则衰减越快
- **amplitude** –放大系数

指数衰减的脉冲-电流转换器

若当前时刻到达一个脉冲，则电流变为 amplitude

否则电流按时间常数为 tau 进行指数衰减

forward(*in_spike*)

参数 **in_spike** –输入脉冲

返回 输出电流

reset()

返回 None

重置所有状态变量为初始值，对于 ExpDecayCurrent 而言，直接将电流设置为 0 即可

training: bool

class SpikingFlow.connection.transform.**STPTransformer**(*v_base*, *tau_f*, *tau_d*)

基类：*SpikingFlow.connection.transform.BaseTransformer*

参数

- **v_base** –v 的基本值
- **tau_f** –刺激信号衰减的时间常数
- **tau_d** –抑制信号衰减的时间常数

突触的短期可塑性。工作在突触前脉冲的时刻，用于调制突触前脉冲的刺激值，使其不至于产生大量突触后电流。

其动态方程为

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{1-x}{\tau_d} - vx\delta(t)$$
$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{V_{base}-v}{\tau_f} + V_{base}(1-v)\delta(t)$$

输出电流为 $vx\delta(t)$

forward(*in_spike*)

参数 **in_spike** –输入脉冲

返回 输出电流

reset()

返回 None

重置所有状态变量 x, v 为初始值 1.0 和 v_base

training: bool

Module contents

class SpikingFlow.connection.**BaseConnection**

基类: torch.nn.modules.module.Module

所有突触的基类

突触，输入和输出均为电流，将脉冲转换为电流的转换器定义在 connection.transform 中

forward(*x*)

参数 **x** –输入电流

返回 输出电流

reset()

返回 None

将突触内的所有状态变量重置为初始状态

training: bool

class SpikingFlow.connection.**ConstantDelay**(*delay_time=1*)

基类：*SpikingFlow.connection.BaseConnection*

参数 **delay_time** –int，表示延迟时长

具有固定延迟 delay_time 的突触，t 时刻的输入，在 t+1+delay_time 时刻才能输出

forward(*x*)

参数 **x** –输入电流

返回 输出电流

t 时刻的输入，在 t+1+delay_time 时刻才能输出

reset()

返回 None

重置状态变量为初始值，对于 ConstantDelay，将保存之前时刻输入的队列清空即可

training: bool

class SpikingFlow.connection.**Linear**(*in_num*, *out_num*, *device='cpu'*)

基类：*SpikingFlow.connection.BaseConnection*

参数

- **in_num** –输入数量
- **out_num** –输出数量
- **device** –数据所在设备

线性全连接层，输入是 [batch_size, *, in_num]，输出是 [batch_size, *, out_num]

连接权重矩阵为 W，输入为 x，输出为 y，则

$$y = xW^T$$

forward(*x*)

参数 **x** –输入电流，shape=[batch_size, *, in_num]

返回 输出电流，shape=[batch_size, *, out_num]

training: bool

class SpikingFlow.connection.**GaussianLinear**(*in_num*, *out_num*, *std*, *device='cpu'*)

基类：*SpikingFlow.connection.BaseConnection*

参数

- **in_num** –输入数量
- **out_num** –输出数量

- **std** –噪声的标准差
- **device** –数据所在设备

带高斯噪声的线性全连接层，噪声是施加在输出端的，所以可以对不同的神经元产生不同的随机噪声。维度上，输入是 [batch_size, *, in_num]，输出是 [batch_size, *, out_num]。

连接权重矩阵为 W，输入为 x，输出为 y，标准差为 std 的噪声为 e, 则

$$y = xW^T + e$$

forward(x)

training: bool

3.1.1.1.2 SpikingFlow.encoding package

Module contents

class SpikingFlow.encoding.**BaseEncoder**

基类: torch.nn.modules.module.Module

所有编码器的基类

编码器将输入数据（例如图像）编码为脉冲数据

forward(x)

参数 **x** –要编码的数据

返回 编码后的脉冲，或者是 None

将 x 编码为脉冲。少数编码器（例如 ConstantEncoder）可以将 x 编码成时长为 1 个 dt 的脉冲，在这种情况下，本函数返回编码后的脉冲

多数编码器（例如 PeriodicEncoder），都是把 x 编码成时长为 n 个 dt 的脉冲 out_spike，out_spike.shape=[n, *]

因此编码一次后，需要调用 n 次 step() 函数才能将脉冲全部发放完毕

第 index 调用 step() 会得到 out_spike[index]

step()

返回 1 个 dt 的脉冲

多数编码器（例如 PeriodicEncoder），编码一次 x，需要经过多步仿真才能将数据输出，这种情况下则用 step 来获取每一步的数据

reset()

返回 None

将编码器的所有状态变量设置为初始状态。对于有状态的编码器，需要重写这个函数

training: bool

class SpikingFlow.encoding.**ConstantEncoder**

基类：*SpikingFlow.encoding.BaseEncoder*

将输入简单转化为脉冲，输入中大于 0 的位置输出 1，其他位置输出 0

forward(*x: torch.Tensor*)

参数 **x** –任意的 tensor

返回 x.bool()，大于 0 的位置为 1，其他位置为 0

training: bool

class SpikingFlow.encoding.**PeriodicEncoder**(*out_spike*)

基类：*SpikingFlow.encoding.BaseEncoder*

参数 **out_spike** –shape=[T, *]，PeriodicEncoder 会不断的输出 out_spike[0], out_spike[1], ⋯, out_spike[T-1], out_spike[0], out_spike[1], ⋯

给定 out_spike 后，周期性的输出 out_spike[0], out_spike[1], ⋯, out_spike[T-1] 的编码器

forward(*x*)

参数 **x** –输入数据，实际上并不需要输入数据，因为 out_spike 在初始化时已经被指定了

返回 调用 step() 后得到的返回值

step()

返回 out_spike[index]

初始化时 index=0，每调用一次，index 则自增 1，index 为 T 时修改为 0

set_out_spike(*out_spike*)

参数 **out_spike** –新设定的 out_spike，必须是 torch.bool

返回 None

重新设定编码器的输出脉冲 self.out_spike 为 out_spike

reset()

返回 None

重置编码器的状态变量，对于 PeriodicEncoder 而言将索引 index 置 0 即可

training: bool

class SpikingFlow.encoding.**LatencyEncoder**(*max_spike_time*, *function_type='linear'*, *device='cpu'*)

基类：*SpikingFlow.encoding.BaseEncoder*

参数

- **max_spike_time** –最晚（最大）脉冲发放时间
- **function_type** –'linear'或'log'
- **device** –数据所在设备

延迟编码，刺激强度越大，脉冲发放越早。要求刺激强度已经被归一化到 [0, 1]

脉冲发放时间 t_i 与刺激强度 x_i 满足

type='linear'

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

type='log'

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

$alpha$ 满足

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

这导致此编码器很容易发生溢出，因为

$$alpha = exp(t_{max} - 1) - 1$$

当 t_{max} 较大时 $alpha$ 极大

示例代码

```
x = torch.rand(size=[3, 2])
max_spike_time = 20
le = encoding.LatencyEncoder(max_spike_time)

le(x)
print(x)
print(le.spike_time)
for i in range(max_spike_time):
    print(le.step())
```

forward(*x*)

参数 **x** –要编码的数据，任意形状的 tensor，要求 x 的数据范围必须在 [0, 1]

将输入数据 x 编码为 max_spike_time 个时刻的 max_spike_time 个脉冲

step()

返回 out_spike[index]

初始化时 index=0，每调用一次，index 则自增 1，index 为 max_spike_time 时修改为 0

reset()

返回 None

重置 LatencyEncoder 的所有状态变量（包括 spike_time，out_spike，index）为初始值 0

training: bool

class SpikingFlow.encoding.**PoissonEncoder**

基类：*SpikingFlow.encoding.BaseEncoder*

泊松频率编码，输出脉冲可以看作是泊松流，发放脉冲的概率即为刺激强度，要求刺激强度已经被归一化到 [0, 1]

示例代码

```
pe = encoding.PoissonEncoder()
x = torch.rand(size=[8])
print(x)
for i in range(10):
    print(pe(x))
```

forward(*x*)

参数 **x** –要编码的数据，任意形状的 tensor，要求 x 的数据范围必须在 [0, 1]

将输入数据 x 编码为脉冲，脉冲发放的概率即为对应位置元素的值

training: bool

class SpikingFlow.encoding.**GaussianTuningCurveEncoder**(*x_min*, *x_max*, *tuning_curve_num*, *max_spike_time*, *device='cpu'*)

基类：*SpikingFlow.encoding.BaseEncoder*

参数

- **x_min** –float，或者是 shape=[M] 的 tensor，表示 M 个特征的最小值
- **x_max** –float，或者是 shape=[M] 的 tensor，表示 M 个特征的最大值
- **tuning_curve_num** –编码每个特征使用的高斯函数（调谐曲线）数量
- **max_spike_time** –最大脉冲发放时间，所有数据都会被编码到 [0, max_spike_time - 1] 范围内的脉冲发放时间
- **device** –数据所在设备

Bohte S M, Kok J N, La Poutre H. Error-backpropagation in temporally encoded networks of spiking neurons[J]. Neurocomputing, 2002, 48(1-4): 17-37.

高斯调谐曲线编码，一种时域编码方法

首先生成 tuning_curve_num 个高斯函数，这些高斯函数的对称轴在数据范围内均匀排列对于每一个输入 x，计算 tuning_curve_num 个高斯函数的值，使用这些函数值线性地生成 tuning_curve_num 个脉冲发放时间

待编码向量是 M 维 tensor，也就是有 M 个特征

1 个 M 维 tensor 会被编码成 shape=[M, tuning_curve_num] 的 tensor，表示 M * tuning_curve_num 个神经元的脉冲发放时间

需要注意的是，编码一次数据，经过 max_spike_time 步仿真，才能进行下一次的编码

示例代码

```
x = torch.rand(size=[3, 2])
tuning_curve_num = 10
max_spike_time = 20
ge = encoding.GaussianTuningCurveEncoder(x.min(0)[0], x.max(0)[0], tuning_curve_
↪num=tuning_curve_num, max_spike_time=max_spike_time)
ge(x)
for i in range(max_spike_time):
    print(ge.step())
```

forward(*x*)

参数 **x** –要编码的数据，shape=[batch_size, M]

将输入数据 x 编码为脉冲

step()

返回 out_spike[index]

初始化时 index=0，每调用一次，index 则自增 1，index 为 max_spike_time 时修改为 0

reset()

返回 None

重置 GaussianTuningCurveEncoder 的所有状态变量（包括 spike_time，out_spike，index）为初始值 0

training: bool

3.1.1.1.3 SpikingFlow.event_driven package

Subpackages

SpikingFlow.event_driven.examples package

Submodules

SpikingFlow.event_driven.examples.tempotron-mnist module

Module contents

Submodules

SpikingFlow.event_driven.encoding module

class SpikingFlow.event_driven.encoding.**GaussianTuning**(*n*, *m*, *x_min: torch.Tensor*, *x_max: torch.Tensor*)

基类：object

参数

- **n** –特征的数量，int
- **m** –编码一个特征所使用的神经元数量，int
- **x_min** –n 个特征的最小值，shape=[n] 的 tensor
- **x_max** –n 个特征的最大值，shape=[n] 的 tensor

Bohte S M, Kok J N, La Poutre J A, et al. Error-backpropagation in temporally encoded networks of spiking neurons[J]. Neurocomputing, 2002, 48(1): 17-37. 中提出的高斯调谐曲线编码方式

编码器所使用的变量所在的 device 与 x_min.device 一致

encode(*x: torch.Tensor*, *max_spike_time=50*)

参数

- **x** –shape=[batch_size, n, k]，batch_size 个数据，每个数据含有 n 个特征，每个特征中有 k 个数据
- **max_spike_time** –最大（最晚）脉冲发放时间，也可以称为编码时间窗口的长度

返回 out_spikes, shape=[batch_size, n, k, m]，将每个数据编码成了 m 个神经元的脉冲发放时间

SpikingFlow.event_driven.neuron module

class SpikingFlow.event_driven.neuron.**Tempotron**(*in_features*, *out_features*, *T*, *tau=15.0*, *tau_s=3.75*, *v_threshold=1.0*)

基类：torch.nn.modules.module.Module

参数

- **in_features** –输入数量，含义与 nn.Linear 的 in_features 参数相同
- **out_features** –输出数量，含义与 nn.Linear 的 out_features 参数相同
- **T** –仿真周期

- **tau** –LIF 神经元的积分时间常数
- **tau_s** –突触上的电流的衰减时间常数
- **v_threshold** –阈值电压

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

static psp_kernel(*t: torch.Tensor*, *tau*, *tau_s*)

参数

- **t** –表示时刻的 tensor
- **tau** –LIF 神经元的积分时间常数
- **tau_s** –突触上的电流的衰减时间常数

返回 t 时刻突触后的 LIF 神经元的电压值

static mse_loss(*v_max*, *v_threshold*, *label*, *num_classes*)

参数

- **v_max** –Tempotron 神经元在仿真周期内输出的最大电压值，与 forward 函数在 ret_type == ‘v_max’时的返回值相同。shape=[batch_size, out_features] 的 tensor
- **v_threshold** –Tempotron 的阈值电压，float 或 shape=[batch_size, out_features] 的 tensor
- **label** –样本的真实标签，shape=[batch_size] 的 tensor
- **num_classes** –样本的类别总数，int

返回 分类错误的神经元的电压，与阈值电压之差的均方误差

forward(*in_spikes: torch.Tensor*, *ret_type*)

参数 **in_spikes** –shape=[batch_size, in_features]

in_spikes[:, i] 表示第 i 个输入脉冲的脉冲发放时刻，介于 0 到 T 之间，T 是仿真时长

in_spikes[:, i] < 0 则表示无脉冲发放:param ret_type: 返回值的类项，可以为’v’,’v_max’,’spikes’ :return:

ret_type == ‘v’ : 返回一个 shape=[batch_size, out_features, T] 的 tensor , 表示 out_features 个 Tempotron 神经元在仿真时长 T 内的电压值

ret_type == ‘v_max’ : 返回一个 shape=[batch_size, out_features] 的 tensor，表示 out_features 个 Tempotron 神经元在仿真时长 T 内的峰值电压

ret_type == ‘spikes’ : 返回一个 out_spikes , shape=[batch_size, out_features] 的 tensor , 表示 out_features 个 Tempotron 神经元的脉冲发放时刻，out_spikes[:, i] 表示第 i 个输出脉冲的脉冲发放时刻，介于 0 到 T 之间，T 是仿真时长。out_spikes[:, i] < 0 表示无脉冲发放

training: bool

Module contents

3.1.1.1.4 SpikingFlow.examples package

Module contents

3.1.1.1.5 SpikingFlow.learning package

Module contents

class SpikingFlow.learning.**STDPModule**(*tf_module*, *connection_module*, *neuron_module*, *tau_pre*, *tau_post*, *learning_rate*, *f_w=<function STDPModule.<lambda>*)

基类：torch.nn.modules.module.Module

参数

- **tf_module** –connection.transform 中的脉冲-电流转换器
- **connection_module** –突触
- **neuron_module** –神经元
- **tau_pre** –pre 脉冲的迹的时间常数
- **tau_post** –post 脉冲的迹的时间常数
- **learning_rate** –学习率
- **f_w** –权值函数，输入是权重 w，输出一个 float 或者是与权重 w 相同 shape 的 tensor

由 tf_module，connection_module，neuron_module 构成的 STDP 学习的基本单元

利用迹的方式（Morrison A, Diesmann M, Gerstner W. Phenomenological models of synaptic plasticity based on spike timing[J]. Biological cybernetics, 2008, 98(6): 459-478.）实现 STDP 学习，更新 connection_module 中的参数

pre 脉冲到达时，权重减少 trace_post * f_w(w) * learning_rate

post 脉冲到达时，权重增加 trace_pre * f_w(w) * learning_rate

示例代码

```
import SpikingFlow.simulating as simulating
import SpikingFlow.learning as learning
import SpikingFlow.connection as connection
import SpikingFlow.connection.transform as tf
```

(下页继续)

(续上页)

```
import SpikingFlow.neuron as neuron
import torch
from matplotlib import pyplot

# 新建一个仿真器
sim = simulating.Simulator()

# 添加各个模块。为了更明显的观察到脉冲，我们使用 IF 神经元，而且把膜电阻设置的很大
# 突触的 pre 是 2 个输入，而 post 是 1 个输出，连接权重是 shape=[1, 2] 的 tensor
sim.append(learning.STDPModule(tf.SpikeCurrent(amplitude=0.5),
                               connection.Linear(2, 1),
                               neuron.IFNode(shape=[1], r=50.0, v_threshold=1.0),
                               tau_pre=10.0,
                               tau_post=10.0,
                               learning_rate=1e-3
                               ))
# 新建 list，分别保存 pre 的 2 个输入脉冲、post 的 1 个输出脉冲，以及对应的连接权重
pre_spike_list0 = []
pre_spike_list1 = []
post_spike_list = []
w_list0 = []
w_list1 = []
T = 200

for t in range(T):
    if t < 100:
        # 前 100 步仿真，pre_spike[0] 和 pre_spike[1] 都是发放一次 1 再发放一次 0
        if t % 2 == 0:
            pre_spike = torch.ones(size=[2], dtype=torch.bool)
        else:
            pre_spike = torch.zeros(size=[2], dtype=torch.bool)
    else:
        # 后 100 步仿真，pre_spike[0] 一直为 0，而 pre_spike[1] 一直为 1
        pre_spike = torch.zeros(size=[2], dtype=torch.bool)
        pre_spike[1] = True

    post_spike = sim.step(pre_spike)
    pre_spike_list0.append(pre_spike[0].float().item())
    pre_spike_list1.append(pre_spike[1].float().item())

    post_spike_list.append(post_spike.float().item())

    w_list0.append(sim.module_list[-1].module_list[2].w[:, 0].item())
```

(下页继续)

(续上页)

```
    w_list1.append(sim.module_list[-1].module_list[2].w[:, 1].item())

# 画出 pre_spike[0]
pyplot.bar(torch.arange(0, T).tolist(), pre_spike_list0, width=0.1, label='pre_
↪spike[0]')
pyplot.legend()
pyplot.show()

# 画出 pre_spike[1]
pyplot.bar(torch.arange(0, T).tolist(), pre_spike_list1, width=0.1, label='pre_
↪spike[1]')
pyplot.legend()
pyplot.show()

# 画出 post_spike
pyplot.bar(torch.arange(0, T).tolist(), post_spike_list, width=0.1, label='post_
↪spike')
pyplot.legend()
pyplot.show()

# 画出 2 个输入与 1 个输出的连接权重 w_0 和 w_1
pyplot.plot(w_list0, c='r', label='w[0]')
pyplot.plot(w_list1, c='g', label='w[1]')
pyplot.legend()
pyplot.show()
```

update_param(*pre=True*)

forward(*pre_spike*)

参数 **pre_spike** –输入脉冲

返回 经过本 module 后的输出脉冲

需要注意的时，由于本 module 含有 tf_module, connection_module, neuron_module 三个 module 因此在 t 时刻的输入，到 t+3dt 才能得到其输出

reset()

返回 None

将 module 自身，以及 tf_module, connection_module, neuron_module 的状态变量重置为初始值

training: bool

class SpikingFlow.learning.**STDPUpdater**(*tau_pre, tau_post, learning_rate, f_w=<function STDPUpdater.<lambda>*)

基类：object

参数

- **neuron_module** –神经元
- **tau_pre** –pre 脉冲的迹的时间常数
- **tau_post** –post 脉冲的迹的时间常数
- **learning_rate** –学习率
- **f_w** –权值函数，输入是权重 w，输出一个 float 或者是与权重 w 相同 shape 的 tensor

利用迹的方式（Morrison A, Diesmann M, Gerstner W. Phenomenological models of synaptic plasticity based on spike timing[J]. Biological cybernetics, 2008, 98(6): 459-478.）实现 STDP 学习，更新 connection_module 中的参数

pre 脉冲到达时，权重减少 trace_post * f_w(w) * learning_rate

post 脉冲到达时，权重增加 trace_pre * f_w(w) * learning_rate

与 STDPModule 类似，但需要手动给定前后脉冲，这也带来了更为灵活的使用方式，例如不使用突触实际连接的前后神经元的脉冲，而是使用其他脉冲来指导某个突触的学习

示例代码

```
import SpikingFlow.simulating as simulating
import SpikingFlow.learning as learning
import SpikingFlow.connection as connection
import SpikingFlow.connection.transform as tf
import SpikingFlow.neuron as neuron
import torch
from matplotlib import pyplot

# 定义权值函数 f_w
def f_w(x: torch.Tensor):
    x_abs = x.abs()
    return x_abs / (x_abs.sum() + 1e-6)

# 新建一个仿真器
sim = simulating.Simulator()

# 放入脉冲电流转换器、突触、LIF 神经元
sim.append(tf.SpikeCurrent(amplitude=0.5))
sim.append(connection.Linear(2, 1))
sim.append(neuron.LIFNode(shape=[1], r=10.0, v_threshold=1.0, tau=100.0))

# 新建一个 STDPUpdater
updater = learning.STDPUpdater(tau_pre=50.0,
                               tau_post=100.0,
```

(下页继续)

(续上页)

```
                                learning_rate=1e-1,
                                f_w=f_w)

# 新建 list, 保存 pre 脉冲、post 脉冲、突触权重 w_00, w_01
pre_spike_list0 = []
pre_spike_list1 = []
post_spike_list = []
w_list0 = []
w_list1 = []

T = 500
for t in range(T):
    if t < 250:
        if t % 2 == 0:
            pre_spike = torch.ones(size=[2], dtype=torch.bool)
        else:
            pre_spike = torch.randint(low=0, high=2, size=[2]).bool()
    else:
        pre_spike = torch.zeros(size=[2], dtype=torch.bool)
        if t % 2 == 0:
            pre_spike[1] = True




    pre_spike_list0.append(pre_spike[0].float().item())
    pre_spike_list1.append(pre_spike[1].float().item())

    post_spike = sim.step(pre_spike)

    updater.update(sim.module_list[1], pre_spike, post_spike)

    post_spike_list.append(post_spike.float().item())

    w_list0.append(sim.module_list[1].w[:, 0].item())
    w_list1.append(sim.module_list[1].w[:, 1].item())

pyplot.figure(figsize=(8, 16))
pyplot.subplot(4, 1, 1)
pyplot.bar(torch.arange(0, T).tolist(), pre_spike_list0, width=0.1, label='pre_
↪spike[0]')
pyplot.legend()
```

(下页继续)

(续上页)

```
pyplot.subplot(4, 1, 2)
pyplot.bar(torch.arange(0, T).tolist(), pre_spike_list1, width=0.1, label='pre_
↪spike[1]')
pyplot.legend()

pyplot.subplot(4, 1, 3)
pyplot.bar(torch.arange(0, T).tolist(), post_spike_list, width=0.1, label='post_
↪spike')
pyplot.legend()

pyplot.subplot(4, 1, 4)
pyplot.plot(w_list0, c='r', label='w[0]')
pyplot.plot(w_list1, c='g', label='w[1]')
pyplot.legend()
pyplot.show()
```

reset()

返回 None

将前后脉冲的迹设置为 0

update(*connection_module*, *pre_spike*, *post_spike*, *inverse=False*)

参数

- **connection_module** –突触
- **pre_spike** –输入脉冲
- **post_spike** –输出脉冲
- **inverse** –为 True 则进行 Anti-STDP

指定突触的前后脉冲，进行 STDP 学习

3.1.1.1.6 SpikingFlow.neuron package

Module contents

class SpikingFlow.neuron.**BaseNode**(*shape*, *r*, *v_threshold*, *v_reset=0.0*, *device='cpu'*)

基类：torch.nn.modules.module.Module

参数

- **shape** –输出的 shape，可以看作是神经元的数量
- **r** –膜电阻，可以是一个 float，表示所有神经元的膜电阻均为这个 float。也可以是形状为 shape 的 tensor，这样就指定了每个神经元的膜电阻

- **v_threshold** –阈值电压，可以是一个 float，也可以是 tensor
- **v_reset** –重置电压，可以是一个 float，也可以是 tensor 注意，更新过程中会确保电压不低于 v_reset，因而电压低于 v_reset 时会被截断为 v_reset
- **device** –数据所在的设备

时钟驱动（逐步仿真）的神经元基本模型

这些神经元都是在 t 时刻接收电流 i 作为输入，与膜电阻 r 相乘，得到 dv = i * r

之后 self.v += dv，然后根据神经元自身的属性，决定是否发放脉冲

需要注意的是，所有的神经元模型都遵循如下约定:

1. 电压会被截断到 [v_reset, v_threshold]
2. 数据经过一个 BaseNode，需要 1 个 dt 的时间。可以参考 simulating 包的流水线的设计
3. t-dt 时刻电压没有达到阈值，t 时刻电压达到了阈值，则到 t+dt 时刻才会放出脉冲。这是为了方便查看波形图，如果不这样设计，若 t-dt 时刻电压为 0.1，v_threshold=1.0，v_reset=0.0, t 时刻增加了 0.9，直接在 t 时刻发放脉冲，则从波形图上看，电压从 0.1 直接跳变到了 0.0，不利于进行数据分析

forward(*i*)

参数 **i** –当前时刻的输入电流，可以是一个 float，也可以是 tensor

Return:out_spike shape 与 self.shape 相同，输出脉冲

接受电流输入，更新膜电位的电压，并输出脉冲（如果过阈值）

reset()

返回 None

将所有状态变量全部设置为初始值，作为基类即为将膜电位 v 设置为 v_reset

对于子类，如果存在除了 v 以外其他状态量的神经元，应该重写此函数

training: bool

class SpikingFlow.neuron.**IFNode**(*shape*, *r*, *v_threshold*, *v_reset=0.0*, *device='cpu'*)

基类: *SpikingFlow.neuron.BaseNode*

参数

- **shape** –输出的 shape，可以看作是神经元的数量
- **r** –膜电阻，可以是一个 float，表示所有神经元的膜电阻均为这个 float。也可以是形状为 shape 的 tensor，这样就指定了每个神经元的膜电阻
- **v_threshold** –阈值电压，可以是一个 float，也可以是 tensor
- **v_reset** –重置电压，可以是一个 float，也可以是 tensor。注意，更新过程中会确保电压不低于 v_reset，因而电压低于 v_reset 时会被截断为 v_reset

- **device** –数据所在的设备

IF 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

$$\frac{\mathrm{d}V(t)}{\mathrm{d}t} = R_m I(t)$$

电压一旦达到阈值 v_threshold 则下一个时刻放出脉冲，同时电压归位到重置电压 v_reset

测试代码

```
if_node = neuron.IFNode([1], r=1.0, v_threshold=1.0)
v = []
for i in range(1000):
    if_node(0.01)
    v.append(if_node.v.item())

pyplot.plot(v)
pyplot.show()
```

forward(*i*)

参数 **i** –当前时刻的输入电流，可以是一个 float，也可以是 tensor

返回 out_spike: shape 与 self.shape 相同，输出脉冲

training: bool

class SpikingFlow.neuron.**LIFNode**(*shape*, *r*, *v_threshold*, *v_reset=0.0*, *tau=1.0*, *device='cpu'*)

基类：*SpikingFlow.neuron.BaseNode*

参数

- **shape** –输出的 shape，可以看作是神经元的数量
- **r** –膜电阻，可以是一个 float，表示所有神经元的膜电阻均为这个 float。也可以是形状为 shape 的 tensor，这样就指定了每个神经元的膜电阻
- **v_threshold** –阈值电压，可以是一个 float，也可以是 tensor
- **v_reset** –重置电压，可以是一个 float，也可以是 tensor 注意，更新过程中会确保电压不低于 v_reset，因而电压低于 v_reset 时会被截断为 v_reset
- **tau** –膜电位时间常数，越大则充电越慢对于频率编码而言，tau 越大，神经元对“频率”的感知和测量也就越精准在分类任务中，增大 tau 在一定范围内能够显著增加正确率
- **device** –数据所在的设备

LIF 神经元模型，可以看作是带漏电的积分器

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

电压在不为 v_reset 时，会指数衰减

```
v_decay = -(self.v - self.v_reset)
self.v += (self.r * i + v_decay) / self.tau
```

电压一旦达到阈值 v_threshold 则下一个时刻放出脉冲，同时电压归位到重置电压 v_reset

测试代码

```
lif_node = neuron.LIFNode([1], r=9.0, v_threshold=1.0, tau=20.0)
v = []

for i in range(1000):
    if i < 500:
        lif_node(0.1)
    else:
        lif_node(0)
    v.append(lif_node.v.item())

pyplot.plot(v)
pyplot.show()
```

training: bool

forward(*i*)

参数 **i** –当前时刻的输入电流，可以是一个 float，也可以是 tensor

返回 out_spike: shape 与 self.shape 相同，输出脉冲

3.1.1.1.7 SpikingFlow.simulating package

Module contents

class SpikingFlow.simulating.**Simulator**(*fast=True*)

基类: object

参数 **fast** –是否快速仿真（仿真的第一时刻就给出输出）。因为 module_list 长度为 n 时，t=0 时刻的输入在 t=n 时刻才流经 module[n-1] 并输出，也就是需要仿真器运行 n 步后才能得到输出。实际使用时可能不太方便，因此若开启快速仿真，则在仿真器首次运行时，会运行 n 步而不是 1 步，并认为这 n 步的输入都是 input_data

仿真器，内含多个 module 组成的 list

当前时刻启动仿真前，数据和模型如下

x[0] -> module[0] -> x[1] -> module[1] -> ⋯-> x[n-1] -> module[n-1] -> x[n]

pipeline = [x[0], x[1], ⋯, x[n-2], x[n-1], x[n]]

启动仿真后，应该按照如下顺序计算

X[0] = input_data

x[n] = module[n-1](x[n-1])

x[n-1] = module[n-2](x[n-2])

...

x[1] = module[0](x[0])

测试代码

```
sim = simulating.Simulator()
sim.append(encoding.ConstantEncoder())
sim.append(tf.SpikeCurrent(amplitude=0.01))
sim.append(neuron.IFNode(shape=[1], r=0.5, v_threshold=1.0))
sim.append(tf.SpikeCurrent(amplitude=0.4))
sim.append(neuron.IFNode(shape=[1], r=2.0, v_threshold=1.0))
sim.append(tf.ExpDecayCurrent(tau=5.0, amplitude=1.0))
sim.append(neuron.LIFNode(shape=[1], r=5.0, v_threshold=1.0, tau=10.0))
v = []
v.extend(([], [], []))
for i in range(1000):
    if i < 800:
        output_data = sim.step(torch.ones(size=[1], dtype=torch.bool))
    else:
        output_data = sim.step(torch.zeros(size=[1], dtype=torch.bool))


    #print(i, sim.pipeline)
    for j in range(3):
        v[j].append(sim.module_list[2 * j + 2].v.item())

pyplot.plot(v[0])
pyplot.show()
pyplot.plot(v[1])
pyplot.show()
pyplot.plot(v[2])
pyplot.show()
```

append(*new_module*)

参数 **new_module** –新添加的模型

返回 None

向 Simulator 的 module_list 中添加 module

只要是 torch.nn.Module 的子类并定义了 forward() 方法，就可以添加 =

step(*input_data*)

参数 **input_data** –输入数据

返回 输出值

输入数据 input_data，仿真一步

reset()

返回 None

重置仿真器到开始仿真前的状态，已经添加的 module 并不会被清除

3.1.1.1.8 SpikingFlow.softbp package

Subpackages

SpikingFlow.softbp.examples package

Submodules

SpikingFlow.softbp.examples.cifar10 module

class SpikingFlow.softbp.examples.cifar10.**Net**(*tau=100.0*, *v_threshold=1.0*, *v_reset=0.0*)

基类：torch.nn.modules.module.Module

forward(*x*)

reset_()

training: bool

SpikingFlow.softbp.examples.cifar10.**main**()

SpikingFlow.softbp.examples.cifar10cmp module

class SpikingFlow.softbp.examples.cifar10cmp.**Net**(*gpu_list*, *tau=100.0*, *v_threshold=1.0*, *v_reset=0.0*)

基类：*SpikingFlow.softbp.ModelPipeline*

reset_()

forward(*x*, *T*)

training: bool

SpikingFlow.softbp.examples.cifar10cmp.**main**()

SpikingFlow.softbp.examples.cifar10mp module

class SpikingFlow.softbp.examples.cifar10mp.**Net**(*gpu_list*, *tau=100.0*, *v_threshold=1.0*, *v_reset=0.0*)

基类: *SpikingFlow.softbp.ModelPipeline*

reset_()

training: bool

SpikingFlow.softbp.examples.cifar10mp.**main**()

SpikingFlow.softbp.examples.cifar10oll module

class SpikingFlow.softbp.examples.cifar10oll.**Net**(*tau=100.0*, *v_threshold=1.0*, *v_reset=0.0*)

基类: torch.nn.modules.module.Module

forward(*x*)

reset_()

training: bool

SpikingFlow.softbp.examples.cifar10oll.**main**()

SpikingFlow.softbp.examples.mnist module

class SpikingFlow.softbp.examples.mnist.**Net**(*tau=100.0*, *v_threshold=1.0*, *v_reset=0.0*)

基类: torch.nn.modules.module.Module

forward(*x*)

reset_()

training: bool

SpikingFlow.softbp.examples.mnist.**main**()

Module contents

Submodules

SpikingFlow.softbp.accelerating module

class SpikingFlow.softbp.accelerating.**multiply_spike**

基类: torch.autograd.function.Function

static forward(*ctx*, *x: torch.Tensor*, *spike: torch.Tensor*)

static backward(*ctx*, *grad_output: torch.Tensor*)

class SpikingFlow.softbp.accelerating.**add_spike**

基类：torch.autograd.function.Function

static forward(*ctx*, *x: torch.Tensor*, *spike: torch.Tensor*)

static backward(*ctx*, *grad_output: torch.Tensor*)

class SpikingFlow.softbp.accelerating.**subtract_spike**

基类：torch.autograd.function.Function

static forward(*ctx*, *x: torch.Tensor*, *spike: torch.Tensor*)

static backward(*ctx*, *grad_output: torch.Tensor*)

SpikingFlow.softbp.accelerating.**add**(*x: torch.Tensor*, *spike: torch.Tensor*)

参数

- **x** –任意 tensor
- **spike** –脉冲 tensor。要求 spike 中的元素只能为 0 或 1，且 spike.shape 必须与 x.shape 相同

返回 x + spike

针对与脉冲这一特殊的数据类型，进行前反向传播加速并保持数值稳定的加法运算。

SpikingFlow.softbp.accelerating.**sub**(*x: torch.Tensor*, *spike: torch.Tensor*)

参数

- **x** –任意 tensor
- **spike** –脉冲 tensor。要求 spike 中的元素只能为 0 或 1，且 spike.shape 必须与 x.shape 相同

返回 x - spike

针对与脉冲这一特殊的数据类型，进行前反向传播加速并保持数值稳定的减法运算。

SpikingFlow.softbp.accelerating.**mul**(*x: torch.Tensor*, *spike: torch.Tensor*)

参数

- **x** –任意 tensor
- **spike** –脉冲 tensor。要求 spike 中的元素只能为 0 或 1，且 spike.shape 必须与 x.shape 相同

返回 x * spike

针对与脉冲这一特殊的数据类型，进行前反向传播加速并保持数值稳定的乘法运算。

class SpikingFlow.softbp.accelerating.**soft_vlotage_transform_function**

基类：torch.autograd.function.Function

static forward(*ctx*, *v: torch.Tensor*, *spike: torch.Tensor*, *v_threshold: float*)

static backward(*ctx*, *grad_output: torch.Tensor*)

SpikingFlow.softbp.accelerating.**soft_vlotage_transform**(*v: torch.Tensor*, *spike: torch.Tensor*, *v_threshold: float*)

参数

- **v** –重置前电压
- **spike** –释放的脉冲
- **v_threshold** –阈值电压

返回 重置后的电压

根据释放的脉冲，以 soft 方式重置电压，即释放脉冲后，电压会减去阈值：$v = v - s \cdot v_{threshold}$。

该函数针对脉冲数据进行了前反向传播的加速，并能节省内存，且保持数值稳定。

class SpikingFlow.softbp.accelerating.**hard_voltage_transform_function**

基类：torch.autograd.function.Function

static forward(*ctx*, *v: torch.Tensor*, *spike: torch.Tensor*, *v_reset: float*)

static backward(*ctx*, *grad_output: torch.Tensor*)

SpikingFlow.softbp.accelerating.**hard_voltage_transform**(*v: torch.Tensor*, *spike: torch.Tensor*, *v_reset: float*)

参数

- **v** –重置前电压
- **spike** –释放的脉冲
- **v_reset** –重置电压

返回 重置后的电压

根据释放的脉冲，以 hard 方式重置电压，即释放脉冲后，电压会直接置为重置电压：$v = v \cdot (1 - s) + v_{reset} \cdot s$。

该函数针对脉冲数据进行了前反向传播的加速，并能节省内存，且保持数值稳定。

SpikingFlow.softbp.functional module

SpikingFlow.softbp.functional.**reset_net**(*net: torch.nn.modules.module.Module*)

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。

SpikingFlow.softbp.functional.**spike_cluster**(*v: torch.Tensor*, *v_threshold*, *T_in: int*)

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** –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

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

Gu P, Xiao R, Pan G, et al. STCA: Spatio-Temporal Credit Assignment with Delayed Feedback in Deep Spiking Neural Networks[C]. international joint conference on artificial intelligence, 2019: 1366-1372. 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
```

SpikingFlow.softbp.functional.**spike_similar_loss**(*spikes: torch.Tensor*, *labels: torch.Tensor*, *kernel_type='linear'*, *loss_type='mse'*, **args*)

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，labels[i][k] == 0 则表示数据 i 不属于第 k 类，允许多标签
- **kernel_type** –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** –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 表示相似矩阵，sim[i][j] == 1 表示数据 i 与数据 j 相似，sim[i][j] == 0 表示数据 i 与数据 j 不相似。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 表示脉冲相似矩阵，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。

kernel_type == ‘linear’，线性内积，$\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \boldsymbol{x_i}^T \boldsymbol{y_j}$。

kernel_type == ‘sigmoid’，sigmoid 内积，$\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \text{sigmoid}(\alpha \boldsymbol{x_i}^T \boldsymbol{y_j})$，其中 $\alpha = args[0]$。

kernel_type == ‘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 sim_p[i][j] = $\frac{\kappa(\boldsymbol{x_i}, \boldsymbol{y_j})}{||\boldsymbol{x_i}|| \cdot ||\boldsymbol{y_j}||}$。

对于相似的数据，根据输入的 loss_type，返回度量 sim 与 sim_p 差异的损失。

loss_type == ‘mse’ 时，返回 sim 与 sim_p 的均方误差（也就是 l2 误差）。

loss_type == ‘l1’ 时，返回 sim 与 sim_p 的 l1 误差。

loss_type == ‘bce’ 时，返回 sim 与 sim_p 的二值交叉熵误差。

注解: 脉冲向量稀疏、离散，最好先使用高斯核进行平滑，然后再计算相似度。

`SpikingFlow.softbp.functional.`**`kernel_dot_product`**(*x: torch.Tensor, y: torch.Tensor, kernel='linear', *args*)

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** –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 $\boldsymbol{y_j}$，则

kernel == ‘linear’，线性内积，$\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \boldsymbol{x_i}^T \boldsymbol{y_j}$。

kernel == ‘polynomial’，多项式内积，$\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = (\boldsymbol{x_i}^T \boldsymbol{y_j})^d$，其中 $d = args[0]$。

kernel == 'sigmoid'，sigmoid 内积，$\kappa(\boldsymbol{x_i}, \boldsymbol{y_j}) = \text{sigmoid}(\alpha \boldsymbol{x_i}^T \boldsymbol{y_j})$，其中 $\alpha = args[0]$。

kernel == '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 $\sigma = args[0]$。

SpikingFlow.softbp.functional.**set_threshold_margin**(*output_layer: SpikingFlow.softbp.neuron.BaseNode, label_one_hot: torch.Tensor, eval_threshold=1.0, threshold0=0.9, threshold1=1.1*)

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** –输出层神经元在测试（推理）时使用的电压阈值
- **threshold0** –输出层神经元在训练时，负样本的电压阈值
- **threshold1** –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。

SpikingFlow.softbp.functional.**redundant_one_hot**(*labels: torch.Tensor, num_classes: int, n: int*)

参数

- **labels** –shape=[batch_size] 的 tensor，表示 batch_size 个标签
- **num_classes** –int，类别总数
- **n** –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
```

(下页继续)

(续上页)

```
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.]])
```

SpikingFlow.softbp.functional.**first_spike_index**(*spikes: torch.Tensor*)

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

输入任意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]]])
```

SpikingFlow.softbp.layer module

class SpikingFlow.softbp.layer.**NeuNorm**(*in_channels, k=0.9*)

基类：torch.nn.modules.module.Module

> **警告：** 可能是错误的实现。测试的结果表明，增加 NeuNorm 后的收敛速度和正确率反而下降了。

参数

- **in_channels** –输入数据的通道数

- **k** –动量项系数

Wu Y, Deng L, Li G, et al. Direct Training for Spiking Neural Networks: Faster, Larger, Better[C]. national conference on artificial intelligence, 2019, 33(01): 1311-1318. 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, W, H]。

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。

forward(*in_spikes: torch.Tensor*)

参数 **in_spikes** –来自上一个卷积层的输出脉冲，shape=[batch_size, in_channels, W, H]

返回 正则化后的脉冲，shape=[batch_size, in_channels, W, H]

reset()

返回 None

本层是一个有状态的层。此函数重置本层的状态变量。

training: bool

class SpikingFlow.softbp.layer.**DCT**(*kernel_size*)

基类：torch.nn.modules.module.Module

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。

forward(*x: torch.Tensor*)

参数 **x** –shape=[*, W, H]，*表示任意额外添加的维度

返回 对 x 进行分块 DCT 变换后得到的 tensor

training: bool

class SpikingFlow.softbp.layer.**AXAT**(*in_features, out_features*)

基类：torch.nn.modules.module.Module

参数

- **in_features** –输入数据的最后 2 维的尺寸
- **out_features** –输出数据的最后 2 维的尺寸

对输入数据 X 进行线性变换 AXA^T 的操作。

要求输入数据的 shape=[*, in_features, in_features]，*表示任意额外添加的维度。

将输入的数据看作是批量个 shape=[in_features, in_features] 的矩阵，而 A 是 shape=[out_features, in_features] 的矩阵。

forward(*x: torch.Tensor*)

参数 **x** –输入数据，shape=[*, in_features, in_features]，*表示任意额外添加的维度

返回 输出数据，shape=[*, out_features, out_features]

training: bool

class SpikingFlow.softbp.layer.**Dropout**(*p=0.5*)

基类：torch.nn.modules.module.Module

参数 **p** –设置为 0 的概率

与 torch.nn.Dropout 的操作相同，但是在每一轮的仿真中，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。

torch.nn.Dropout 在 SNN 中使用时，由于 SNN 需要运行一定的步长，每一步运行（t=0,1,⋯,T-1）时都会有不同的 dropout，导致网络的结构实际上是在持续变化：例如可能出现 t=0 时刻，i 到 j 的连接被断开，但 t=1 时刻，i 到 j 的连接又被保持。

在 SNN 中的 dropout 应该是，当前这一轮的运行中，t=0 时若 i 到 j 的连接被断开，则之后 t=1,2,⋯,T-1 时刻，i 到 j 的连接应该一直被断开；而到了下一轮运行时，重新按照概率去决定 i 到 j 的连接是否断开，因此重写了适用于 SNN 的 Dropout。

小技巧: 从之前的实验结果可以看出，当使用 LIF 神经元，损失函数或分类结果被设置成时间上累计输出的值，torch.nn.Dropout 几乎对 SNN 没有影响，即便 dropout 的概率被设置成高达 0.9。可能是 LIF 神经元的积分行为，对某一个时刻输入的缺失并不敏感。

---

forward(*x: torch.Tensor*)

参数 **x** –shape=[*] 的 tensor

返回 shape 与 x.shape 相同的 tensor

reset()

返回 None

本层是一个有状态的层。此函数重置本层的状态变量。

training: bool

class SpikingFlow.softbp.layer.**Dropout2d**(*p=0.2*)

基类：torch.nn.modules.module.Module

参数 **p** –设置为 0 的概率

与 torch.nn.Dropout2d 的操作相同，但是在每一轮的仿真中，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。

forward(*x: torch.Tensor*)

参数 **x** –shape=[N, C, W, H] 的 tensor

返回 shape=[N, C, W, H]，与 x.shape 相同的 tensor

reset()

返回 None

本层是一个有状态的层。此函数重置本层的状态变量。

training: bool

class SpikingFlow.softbp.layer.**LowPassSynapse**(*tau=100.0*, *learnable=False*)

基类：torch.nn.modules.module.Module

参数

- **tau** –突触上电流衰减的时间常数
- **learnable** –时间常数是否设置成可以学习的参数。当设置为可学习参数时，函数参数中的 tau 是该参数的初始值

具有低通滤波性质的突触。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：

$$I(t) = I(t-1) - (1 - S(t))\frac{1}{\tau}I(t-1) + S(t)$$

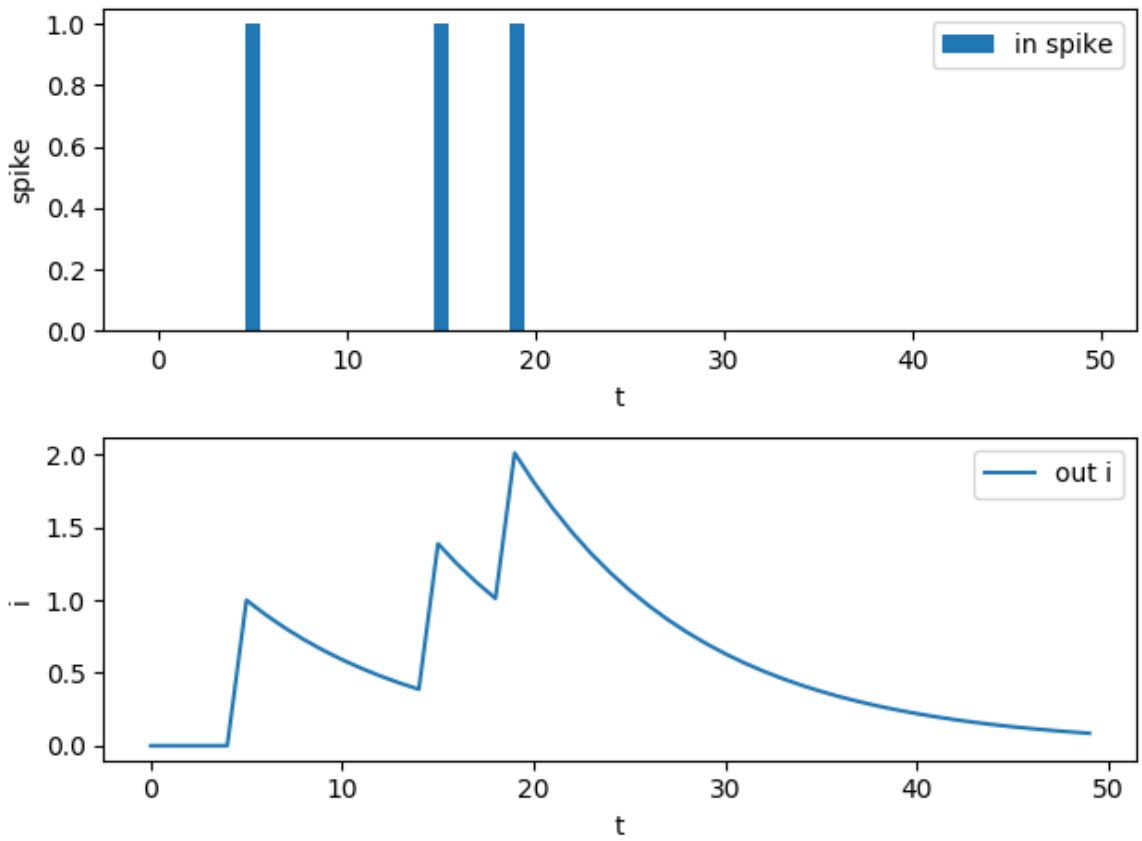
这种突触能将输入脉冲进行“平滑”，简单的示例代码和输出结果：

```
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()
```

(下页继续)

(续上页)

```
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：

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

Fang H, Shrestha A, Zhao Z, et al. Exploiting Neuron and Synapse Filter Dynamics in Spatial Temporal Learning of Deep Spiking Neural Network[J]. arXiv: Neural and Evolutionary Computing, 2020.

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：

Lee C, Sarwar S S, Panda P, et al. Enabling spike-based backpropagation for training deep neural network architectures[J]. Frontiers in Neuroscience, 2020, 14.

forward(*in_spikes: torch.Tensor*)

参数 **in_spikes** –shape 任意的输入脉冲

返回 shape 与 in_spikes.shape 相同的输出电流

reset()

返回 None

本层是一个有状态的层。此函数重置本层的状态变量。将电流重置为 0。

training: bool

class SpikingFlow.softbp.layer.**ChannelsMaxPool**(*pool: torch.nn.modules.pooling.MaxPool1d*)

基类：torch.nn.modules.module.Module

参数 **pool** –nn.Maxpool1d 的池化层

在通道所在的维度，第 1 维，进行池化的层。

forward(*x: torch.Tensor*)

参数 **x** –shape=[batch_size, C_in, *] 的 tensor，C_in 是输入数据的通道数，*表示任意维度

返回 shape=[batch_size, C_out, *] 的 tensor，C_out 是池化后的通道数

training: bool

SpikingFlow.softbp.neuron module

class SpikingFlow.softbp.neuron.**BaseNode**(*v_threshold=1.0, v_reset=0.0, pulse_soft=Sigmoid(), monitor_state=False*)

基类：torch.nn.modules.module.Module

参数

- **v_threshold** –神经元的阈值电压
- **v_reset** –神经元的重置电压。如果不为 None，当神经元释放脉冲后，电压会被重置为 v_reset；如果设置为 None，则电压会被减去阈值
- **pulse_soft** –反向传播时用来计算脉冲函数梯度的替代函数，即软脉冲函数
- **monitor_state** –是否设置监视器来保存神经元的电压和释放的脉冲。若为 True，则 self.monitor 是一个字典，键包括’v’和’s’，分别记录电压和输出脉冲。对应的

值是一个链表。为了节省显存（内存），列表中存入的是原始变量转换为 numpy 数组后的值。还需要注意，self.reset() 函数会清空这些链表

softbp 包中，可微分 SNN 神经元的基类神经元。

可微分 SNN 神经元，在前向传播时输出真正的脉冲（离散的 0 和 1）。脉冲的产生过程可以看作是一个阶跃函数：

$$S = \Theta(V - V_{threshold})$$

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

$\Theta(x)$ 是一个不可微的函数，用一个形状与其相似的函数 $\sigma(x)$，即代码中的 pulse_soft 去近似它的梯度。默认的 pulse_soft = SpikingFlow.softbp.soft_pulse_function.Sigmoid()，在反向传播时用 $\sigma'(x)$ 来近似 $\Theta'(x)$，这样就可以使用梯度下降法来更新 SNN 了。

前向传播使用 $\Theta(x)$，反向传播时按前向传播为 $\sigma(x)$ 来计算梯度，在 PyTorch 中很容易实现，参见这个类的 spiking() 函数。

set_monitor(*monitor_state=True*)

参数 **monitor_state** –True 或 False，表示开启或关闭 monitor

返回 None

设置开启或关闭 monitor。

spiking()

返回 神经元的输出脉冲

根据当前神经元的电压、阈值、重置电压，计算输出脉冲，并更新神经元的电压。

forward(*dv: torch.Tensor*)

参数 **dv** –输入到神经元的电压增量

返回 神经元的输出脉冲

子类需要实现这一函数。

reset()

返回 None

重置神经元为初始状态，也就是将电压设置为 v_reset。

如果子类的神经元还含有其他状态变量，需要在此函数中将这些状态变量全部重置。

training: bool

class SpikingFlow.softbp.neuron.**IFNode**(*v_threshold=1.0, v_reset=0.0, pulse_soft=Sigmoid(), monitor_state=False*)

基类：*SpikingFlow.softbp.neuron.BaseNode*

参数

- **v_threshold** –神经元的阈值电压
- **v_reset** –神经元的重置电压
- **pulse_soft** –反向传播时用来计算脉冲函数梯度的替代函数，即软脉冲函数
- **monitor_state** –是否设置监视器来保存神经元的电压和释放的脉冲。若为 True，则 self.monitor 是一个字典，键包括’v’和’s’，分别记录电压和输出脉冲。对应的值是一个链表。为了节省显存（内存），列表中存入的是原始变量转换为 numpy 数组后的值。还需要注意，self.reset() 函数会清空这些链表

IF 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：

$$\frac{\mathrm{d}V(t)}{\mathrm{d}t} = R_m I(t)$$

电压一旦达到阈值 v_threshold 则放出脉冲，同时电压归位到重置电压 v_reset。

forward(*dv: torch.Tensor*)

training: bool

class SpikingFlow.softbp.neuron.**LIFNode**(*tau=100.0*, *v_threshold=1.0*, *v_reset=0.0*, *pulse_soft=Sigmoid()*, *monitor_state=False*)

基类：*SpikingFlow.softbp.neuron.BaseNode*

参数

- **tau** –膜电位时间常数，越大则充电越慢
- **v_threshold** –神经元的阈值电压
- **v_reset** –神经元的重置电压
- **pulse_soft** –反向传播时用来计算脉冲函数梯度的替代函数，即软脉冲函数
- **monitor_state** –是否设置监视器来保存神经元的电压和释放的脉冲。若为 True，则 self.monitor 是一个字典，键包括’v’和’s’，分别记录电压和输出脉冲。对应的值是一个链表。为了节省显存（内存），列表中存入的是原始变量转换为 numpy 数组后的值。还需要注意，self.reset() 函数会清空这些链表

LIF 神经元模型，可以看作是带漏电的积分器：

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

电压在不为 v_reset 时，会指数衰减。

forward(*dv: torch.Tensor*)

training: bool

class SpikingFlow.softbp.neuron.**PLIFNode**(*init_tau=2.0*, *v_threshold=1.0*, *v_reset=0.0*, *pulse_soft=Sigmoid()*, *monitor_state=False*)

基类：*SpikingFlow.softbp.neuron.BaseNode*

参数

- **init_tau** –初始的 tau
- **v_threshold** –神经元的阈值电压
- **v_reset** –神经元的重置电压
- **pulse_soft** –反向传播时用来计算脉冲函数梯度的替代函数，即软脉冲函数
- **monitor** –是否设置监视器来保存神经元的电压和释放的脉冲。若为 True，则 self.monitor 是一个字典，键包括’v’和’s’，分别记录电压和输出脉冲。对应的值是一个链表。为了节省显存（内存），列表中存入的是原始变量转换为 numpy 数组后的值。还需要注意，self.reset() 函数会清空这些链表

Parametric LIF 神经元模型，时间常数 tau 可学习的 LIF 神经元：

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

电压在不为 v_reset 时，会指数衰减。对于同一层神经元，它们的 tau 是共享的。

小技巧： LIF 神经元的电压更新方程为：

```
self.v += (dv - (self.v - self.v_reset)) / self.tau
```

为了防止出现除以 0 的情况，PLIF 神经元没有使用除法，而是用乘法代替：

```
self.v += (dv - (self.v - self.v_reset)) * self.tau
```

---

forward(*dv: torch.Tensor*)

training: bool

class SpikingFlow.softbp.neuron.**RIFNode**(*init_w=- 0.001*, *amplitude=None*, *v_threshold=1.0*, *v_reset=0.0*, *pulse_soft=Sigmoid()*, *monitor_state=False*)

基类：*SpikingFlow.softbp.neuron.BaseNode*

参数

- **init_w** –初始的自连接权重
- **amplitude** –对自连接权重的限制。若为 `None`，则不会对权重有任何限制；若为一个 `float`，会限制权重在 `(- amplitude, amplitude)` 范围内；若为一个 `tuple`，会限制权重在 `(amplitude[0], amplitude[1])` 范围内。
- **v_threshold** –神经元的阈值电压
- **v_reset** –神经元的重置电压
- **pulse_soft** –反向传播时用来计算脉冲函数梯度的替代函数，即软脉冲函数

- **monitor_state** –是否设置监视器来保存神经元的电压和释放的脉冲。若为 True，则 self.monitor 是一个字典，键包括’v’和’s’，分别记录电压和输出脉冲。对应的值是一个链表。为了节省显存（内存），列表中存入的是原始变量转换为 numpy 数组后的值。还需要注意，self.reset() 函数会清空这些链表

Recurrent IF 神经元模型。与 Parametric LIF 神经元模型类似，但有微妙的区别，自连接权重不会作用于输入。其膜电位更新方程为：

$$\frac{\mathrm{d}V(t)}{\mathrm{d}t} = w(V(t) - V_{reset}) + R_m I(t)$$

其中 w 是自连接权重，权重是可以学习的。对于同一层神经元，它们的 w 是共享的。

training: bool

w()

返回 返回自连接权重

forward(*dv: torch.Tensor*)

SpikingFlow.softbp.optim module

class SpikingFlow.softbp.optim.**AdamRewiring**(*params, lr=0.001, betas=0.9, 0.999, eps=1e-08, weight_decay=0, amsgrad=False, T=1e-05, l1=1e-05*)

基类：torch.optim.optimizer.Optimizer

> **注意：** 该算法的收敛性尚未得到任何证明，以及在基于 softbp 的 SNN 上的剪枝可靠性也未知。

参数

- **params** –（原始 Adam）网络参数的迭代器，或者由字典定义的参数组
- **lr** –（原始 Adam）学习率
- **betas** –（原始 Adam）用于计算运行时梯度平均值的以及平均值平方的两个参数
- **eps** –（原始 Adam）除法计算时，加入到分母中的小常数，用于提高数值稳定性
- **weight_decay** –（原始 Adam）L2 范数惩罚因子
- **amsgrad** –（原始 Adam）是否使用 AMSGrad 算法
- **T** –Deep R 算法中的温度参数
- **l1** –Deep R 算法中的 L1 惩罚参数

G. Bellec et al, “Deep Rewiring: Training very sparse deep networks,” ICLR 2018.

该实现将论文中的基于 SGD 优化算法的 Deep R 算法移植到 Adam: A Method for Stochastic Optimization 优化算法上，是基于 Adam 算法在 Pytorch 中的 官方实现 修改而来。

step(*closure=None*)

参数 closure –（原始 Adam）传入的闭包，可用于评估模型并返回损失

执行单步参数更新

SpikingFlow.softbp.soft_pulse_function module

class SpikingFlow.softbp.soft_pulse_function.**bilinear_leaky_relu**

基类: torch.autograd.function.Function

static forward(*ctx*, *x*, *a=1*, *b=0.01*, *c=0.5*)

static backward(*ctx*, *grad_output*)

class SpikingFlow.softbp.soft_pulse_function.**BilinearLeakyReLU**(*a=1*, *b=0.01*, *c=0.5*)

基类: torch.nn.modules.module.Module

参数

- **a** –-c <= x <= c 时反向传播的梯度
- **b** –x > c 或 x < -c 时反向传播的梯度
- **c** –决定梯度区间的参数

返回 与输入相同 shape 的输出

双线性的近似脉冲发放函数。梯度为

$$g'(x) = \begin{cases} a, & -c \leq x \leq c \\ b, & x < -c \ or \ x > c \end{cases}$$

对应的原函数为

$$g(x) = \begin{cases} bx + bc - ac, & x < -c \\ ax, & -c \leq x \leq c \\ bx - bc + ac, & x > c \end{cases}$$

forward(*x*)

training: bool

class SpikingFlow.softbp.soft_pulse_function.**sigmoid**

基类: torch.autograd.function.Function

static forward(*ctx*, *x*, *alpha*)

static backward(*ctx*, *grad_output*)

class SpikingFlow.softbp.soft_pulse_function.**Sigmoid**(*alpha=1.0*)

基类: torch.nn.modules.module.Module

参数

- **x** –输入数据
- **alpha** –控制反向传播时梯度的平滑程度的参数

返回 与输入相同 shape 的输出

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

$$g(x) = \text{sigmoid}(\alpha x) = \frac{1}{1 + e^{-\alpha x}}$$

forward(*x*)

training: bool

class SpikingFlow.softbp.soft_pulse_function.**sign_swish**

基类: torch.autograd.function.Function

static forward(*ctx*, *x*, *beta=1.0*)

static backward(*ctx*, *grad_output*)

class SpikingFlow.softbp.soft_pulse_function.**SignSwish**(*beta=5.0*)

基类: torch.nn.modules.module.Module

参数

- **x** –输入数据
- **beta** –控制反向传播的参数

返回 与输入相同 shape 的输出

Darabi, Sajad, et al. "BNN+: Improved binary network training." arXiv preprint arXiv:1812.11800 (2018).

反向传播时使用 swish 的梯度的脉冲发放函数。反向传播为

$$g'(x) = \frac{\beta(2 - \beta x \tanh\frac{\beta x}{2})}{1 + \cosh(\beta x)}$$

对应的原函数为

$$g(x) = 2 * \text{sigmoid}(\beta x) * (1 + \beta x(1 - \text{sigmoid}(\beta x))) - 1$$

forward(*x*)

training: bool

Module contents

class SpikingFlow.softbp.**ModelPipeline**

基类：torch.nn.modules.module.Module

用于解决显存不足的模型流水线。将一个模型分散到各个 GPU 上，流水线式的进行训练。设计思路与仿真器非常类似。

运行时建议先取一个很小的 batch_size，然后观察各个 GPU 的显存占用，并调整每个 module_list 中包含的模型比例。

append(*nn_module*, *gpu_id*)

参数

- **nn_module** –新添加的 module
- **gpu_id** –该模型所在的 GPU，不需要带“cuda:”的前缀。例如“2”

返回 None

将 nn_module 添加到流水线中，nn_module 会运行在设备 gpu_id 上。添加的 nn_module 会按照它们的添加顺序运行。例如首先添加了 fc1，又添加了 fc2，则实际运行是按照 input_data->fc1->fc2->output_data 的顺序运行。

constant_forward(*x*, *T*, *reduce=True*)

参数

- **x** –输入数据
- **T** –运行时长
- **reduce** –为 True 则返回运行 T 个时长，得到 T 个输出的和；为 False 则返回这 T 个输出

返回 T 个输出的和或 T 个输出

让本模型以恒定输入 x 运行 T 次，这常见于使用频率编码的 SNN。这种方式比 forward(x, split_sizes) 的运行速度要快很多

forward(*x*, *split_sizes*)

参数

- **x** –输入数据
- **split_sizes** –输入数据 x 会在维度 0 上被拆分成每 split_size 一组，得到 [x0, x1, …]，这些数据会被串行的送入 module_list 中的各个模块进行计算

返回 输出数据

例如将模型分成 4 部分，因而 `module_list` 中有 4 个子模型；将输入分割为 3 部分，则每次调用 `forward(x, split_sizes)` ，函数内部的计算过程如下：

```
step=0      x0, x1, x2  |m0|     |m1|     |m2|     |m3|

step=1      x0, x1      |m0| x2 |m1|     |m2|     |m3|

step=2      x0          |m0| x1 |m1| x2 |m2|     |m3|

step=3                  |m0| x0 |m1| x1 |m2| x2 |m3|

step=4                  |m0|     |m1| x0 |m2| x1 |m3| x2

step=5                  |m0|     |m1|     |m2| x0 |m3| x1, x2

step=6                  |m0|     |m1|     |m2|     |m3| x0, x1, x2
```

不使用流水线，则任何时刻只有一个 GPU 在运行，而其他 GPU 则在等待这个 GPU 的数据；而使用流水线，例如上面计算过程中的 `step=3` 到 `step=4`，尽管在代码的写法为顺序执行：

```
x0 = m1(x0)
x1 = m2(x1)
x2 = m3(x2)
```

但由于 PyTorch 优秀的特性，上面的 3 行代码实际上是并行执行的，因为这 3 个在 CUDA 上的计算使用各自的数据，互不影响

training: bool

3.1.1.1.9 SpikingFlow.visualizing package

Module contents

`SpikingFlow.visualizing.`**`plot_2d_heatmap`**(*array: numpy.ndarray, title: str, xlabel: str, ylabel: str, int_x_ticks=True, int_y_ticks=True, plot_colorbar=True, colorbar_y_label='magnitude', dpi=200*)

参数

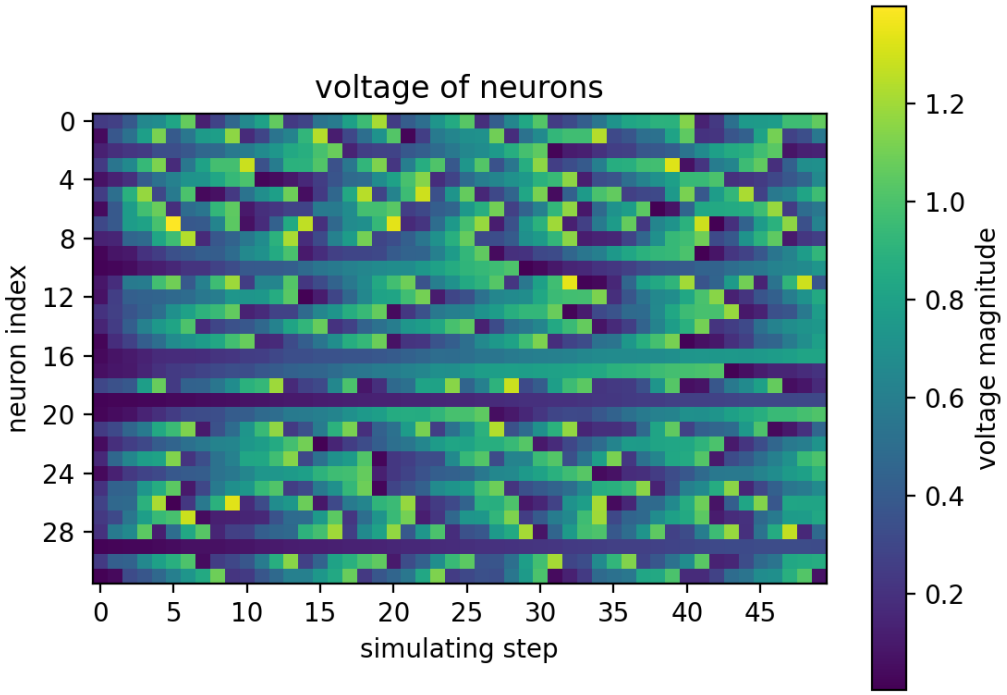
- **array** –shape=[N, M] 的任意数组
- **title** –热力图的标题
- **xlabel** –热力图的 x 轴的 label

- **ylabel** –热力图的 y 轴的 label
- **int_x_ticks** –x 轴上是否只显示整数刻度
- **int_y_ticks** –y 轴上是否只显示整数刻度
- **plot_colorbar** –是否画出显示颜色和数值对应关系的 colorbar
- **colorbar_y_label** –colorbar 的 y 轴 label
- **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：

```
neuron_num = 32
T = 50
lif_node = neuron.LIFNode(monitor=True)
w = torch.rand([neuron_num]) * 50
for t in range(T):
    lif_node(w * torch.rand(size=[neuron_num]))
v_t_array = np.asarray(lif_node.monitor['v']).T  # v_t_array[i][j] 表示神经元 i 在 j
时刻的电压值
visualizing.plot_2d_heatmap(array=v_t_array, title='voltage of neurons', xlabel=
↪'simulating step',
                            ylabel='neuron index', int_x_ticks=True, int_y_
↪ticks=True,
                            plot_colorbar=True, colorbar_y_label='voltage␣
↪magnitude', dpi=200)
plt.show()
```



`SpikingFlow.visualizing.`**`plot_2d_bar_in_3d`**(*array: numpy.ndarray, title: str, xlabel: str, ylabel: str, zlabel: str, int_x_ticks=True, int_y_ticks=True, int_z_ticks=False, dpi=200*)

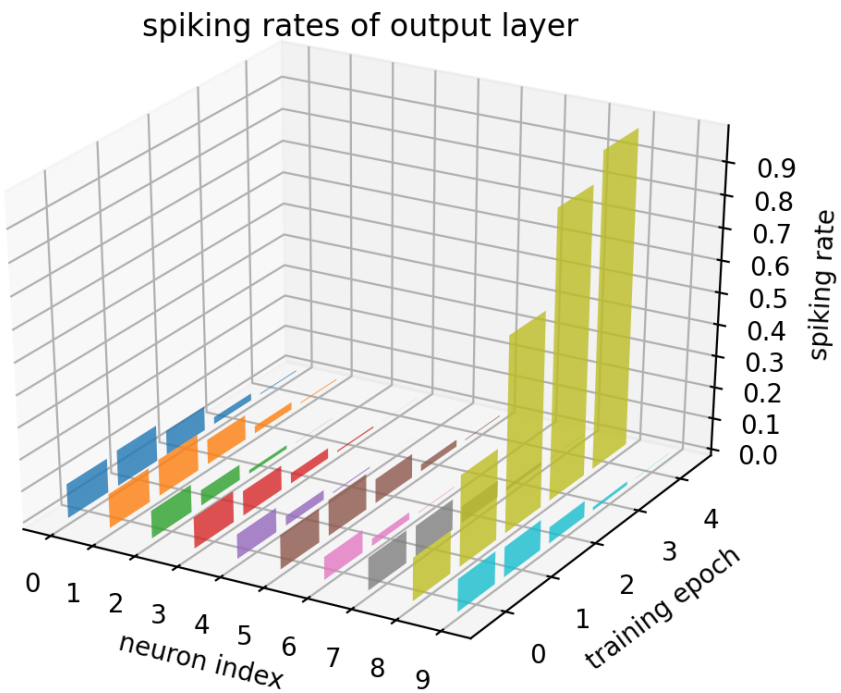
参数

- **array** –shape=[N, M] 的任意数组
- **title** –图的标题
- **xlabel** –x 轴的 label
- **ylabel** –y 轴的 label
- **zlabel** –z 轴的 label
- **int_x_ticks** –x 轴上是否只显示整数刻度
- **int_y_ticks** –y 轴上是否只显示整数刻度
- **int_z_ticks** –z 轴上是否只显示整数刻度
- **dpi** –绘图的 dpi

返回 绘制好的 figure

将 shape=[N, M] 的任意数组，绘制为三维的柱状图。可以用来绘制多个神经元的脉冲发放频率，随着时间的变化情况，示例代码：

```
Epochs = 5
N = 10
spiking_rate = torch.zeros(N, Epochs)
init_spiking_rate = torch.rand(size=[N])
for i in range(Epochs):
    spiking_rate[:, i] = torch.softmax(init_spiking_rate * (i + 1) ** 2, dim=0)
visualizing.plot_2d_bar_in_3d(spiking_rate.numpy(), title='spiking rates of␣
↪output layer', xlabel='neuron index',
                              ylabel='training epoch', zlabel='spiking rate', int_
↪x_ticks=True, int_y_ticks=True,
                              int_z_ticks=False, dpi=200)
plt.show()
```



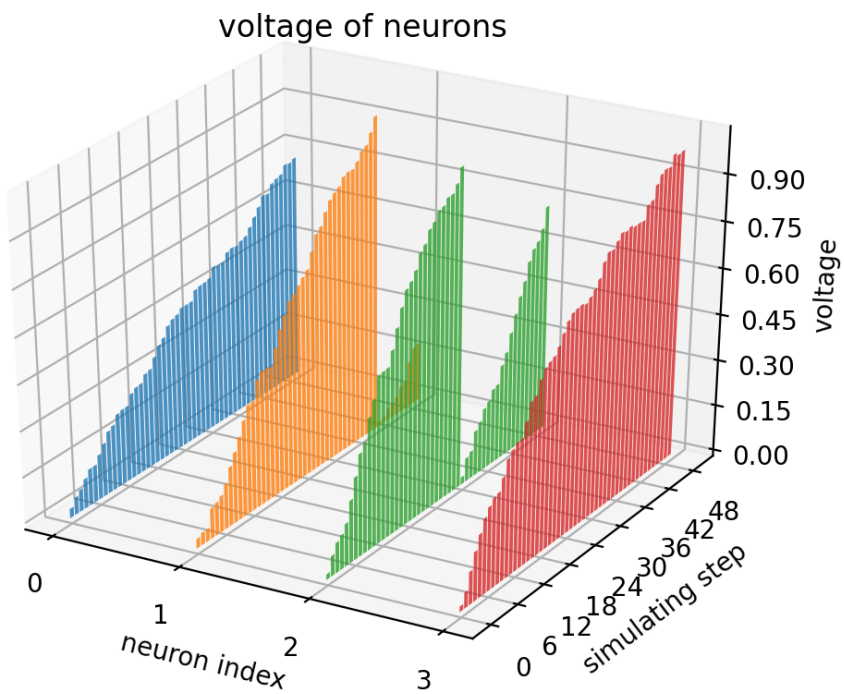
也可以用来绘制一张表示多个神经元在不同时刻的电压的热力图，示例代码：

```
neuron_num = 4
T = 50
lif_node = neuron.LIFNode(monitor=True)
```

(下页继续)

(续上页)

```
w = torch.rand([neuron_num]) * 10
for t in range(T):
    lif_node(w * torch.rand(size=[neuron_num]))
v_t_array = np.asarray(lif_node.monitor['v']).T  # v_t_array[i][j] 表示神经元 i 在 j 时刻的电压值
visualizing.plot_2d_bar_in_3d(v_t_array, title='voltage of neurons', xlabel='neuron index',
                              ylabel='simulating step', zlabel='voltage', int_x_ticks=True, int_y_ticks=True,
                              int_z_ticks=False, dpi=200)
plt.show()
```



SpikingFlow.visualizing.**plot_1d_spikes**(*spikes: numpy.asarray, title: str, xlabel: str, ylabel: str, int_x_ticks=True, int_y_ticks=True, plot_spiking_rate=True, spiking_rate_map_title='spiking rate', dpi=200*)

参数

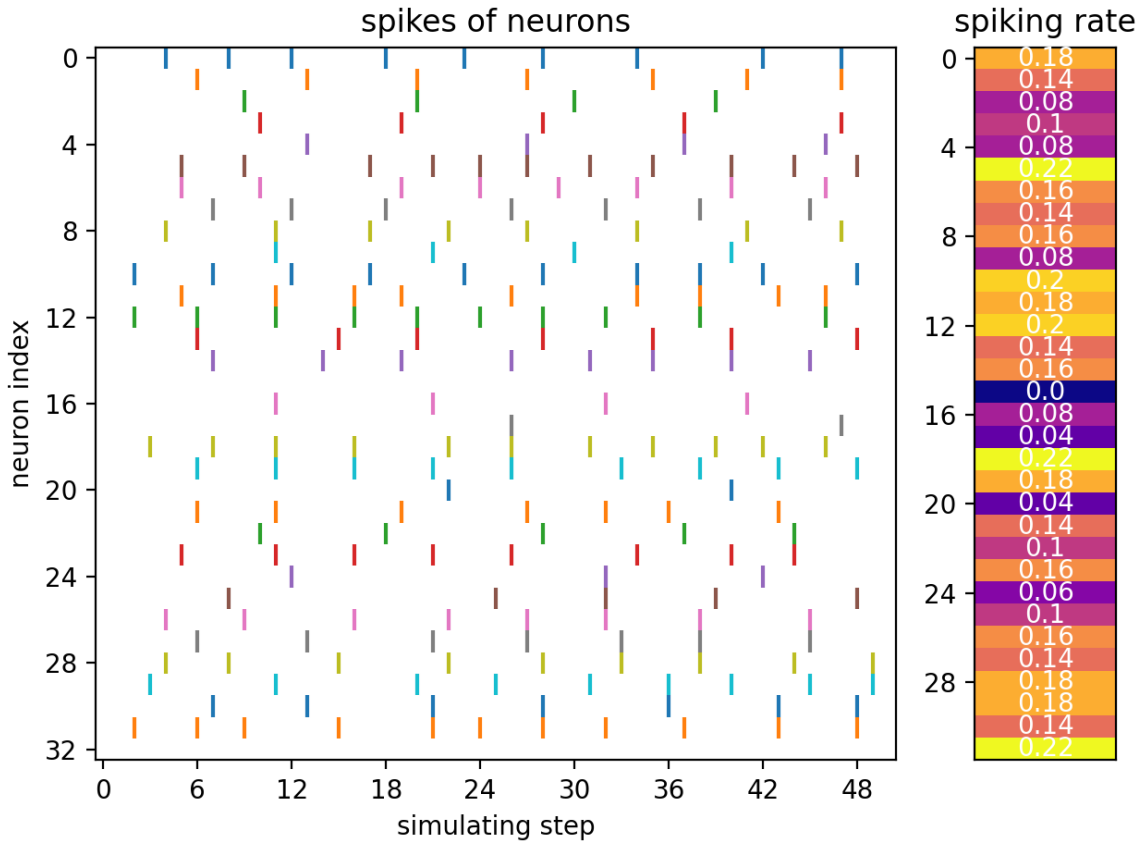
- **spikes** –shape=[N, T] 的 np 数组，其中的元素只为 0 或 1，表示 N 个时长为 T 的脉冲数据

- **title** –热力图的标题
- **xlabel** –热力图的 x 轴的 label
- **ylabel** –热力图的 y 轴的 label
- **int_x_ticks** –x 轴上是否只显示整数刻度
- **int_y_ticks** –y 轴上是否只显示整数刻度
- **plot_spiking_rate** –是否画出各个脉冲发放频率
- **spiking_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：

```
neuron_num = 32
T = 50
lif_node = neuron.LIFNode(monitor=True)
w = torch.rand([neuron_num]) * 50
for t in range(T):
    lif_node(w * torch.rand(size=[neuron_num]))
s_t_array = np.asarray(lif_node.monitor['s']).T  # s_t_array[i][j] 表示神经元 i 在 j
时刻释放的脉冲，为 0 或 1
visualizing.plot_1d_spikes(spikes=s_t_array, title='spikes of neurons', xlabel=
↪'simulating step',
                           ylabel='neuron index', int_x_ticks=True, int_y_
↪ticks=True,
                           plot_spiking_rate=True, spiking_rate_map_title=
↪'spiking rate', dpi=200)
plt.show()
```



SpikingFlow.visualizing.**plot_2d_spiking_feature_map**(*spikes: numpy.asarray*, *nrows*, *ncols*, *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
- **title** –热力图的标题
- **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

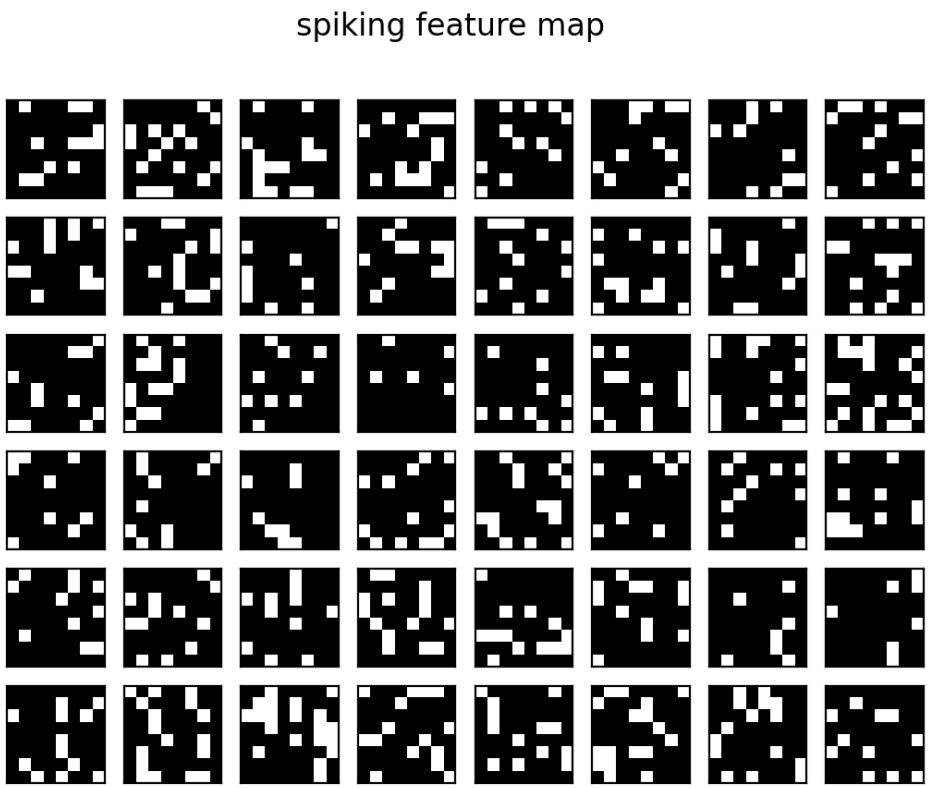
将 C 个尺寸为 W * H 的脉冲矩阵，全部画出，然后排列成 nrows 行 ncols 列。这样的矩阵一般来源于卷积层后的脉冲神经元的输出，通过这个函数可以对输出进行可视化。示例代码：

```
C = 48
W = 8
H = 8
spikes = (np.random.rand(C, W, H) > 0.8).astype(float)
```

(下页继续)

(续上页)

```
visualizing.plot_2d_spiking_feature_map(spikes=spikes, nrows=6, ncols=8, title=
↪'spiking feature map', dpi=200)
plt.show()
```



3.1.1.2 Module contents

3.1.2 快速上手教程

3.1.2.1 神经元 SpikingFlow.neuron

本教程作者：fangwei123456

本节教程主要关注 `SpikingFlow.neuron`，包括如何使用已有神经元、如何定义新的神经元。

3.1.2.1.1 LIF 神经元仿真

我们使用一个 LIF 神经元进行仿真，代码如下：

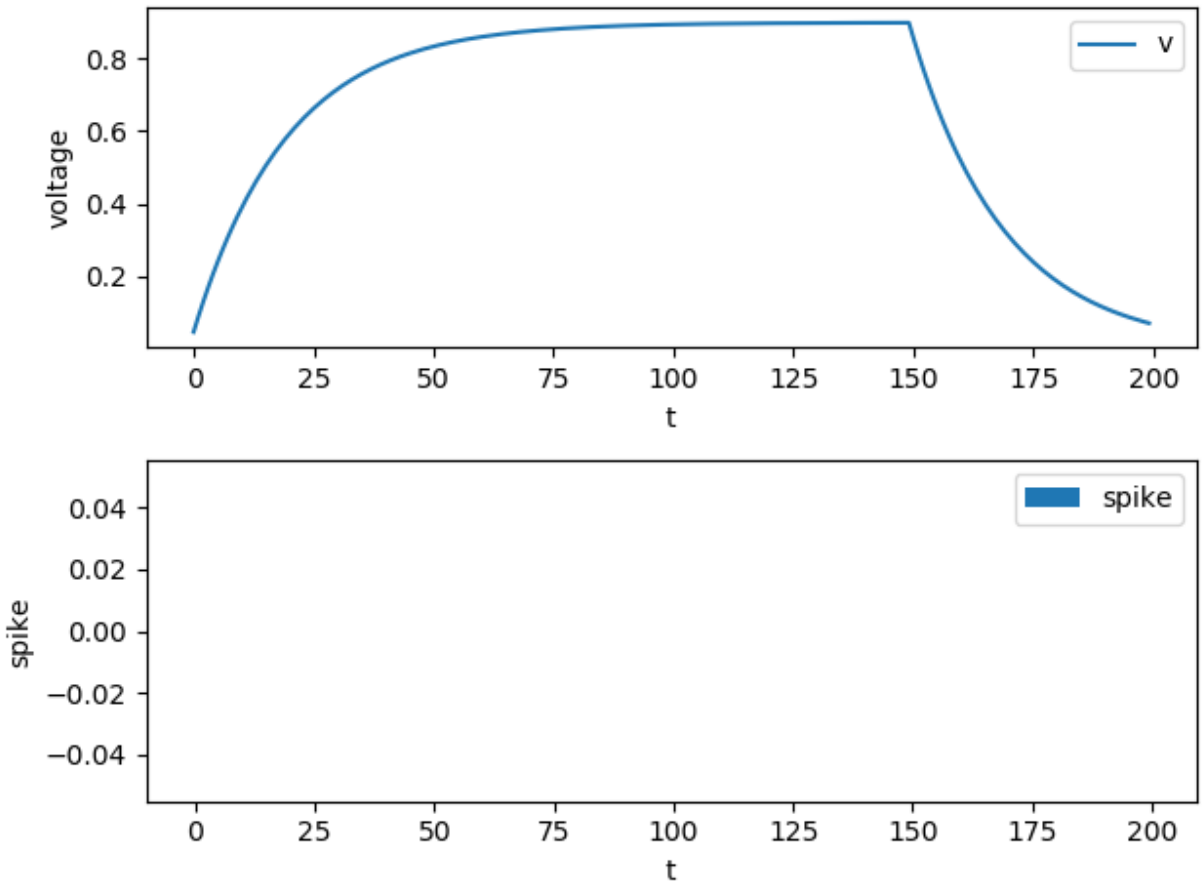
```
import SpikingFlow
import SpikingFlow.neuron as neuron
# 导入绘图模块
from matplotlib import pyplot
import torch

# 新建一个 LIF 神经元
lif_node = neuron.LIFNode([1], r=9.0, v_threshold=1.0, tau=20.0)
# 新建一个空 list，保存仿真过程中神经元的电压值
v_list = []
# 新建一个空 list，保存神经元的输出脉冲
spike_list = []

T = 200
# 运行 200 次
for t in range(T):
    # 前 150 次，输入电流都是 0.1
    if t < 150:
        spike_list.append(lif_node(0.1).float().item())
    # 后 50 次，不输入，也就是输入 0
    else:
        spike_list.append(lif_node(0).float().item())

    # 记录每一次输入后，神经元的电压
    v_list.append(lif_node.v.item())

# 画出电压的变化
pyplot.subplot(2, 1, 1)
pyplot.plot(v_list, label='v')
pyplot.xlabel('t')
pyplot.ylabel('voltage')
pyplot.legend()

# 画出脉冲
pyplot.subplot(2, 1, 2)
pyplot.bar(torch.arange(0, T).tolist(), spike_list, label='spike')
pyplot.xlabel('t')
pyplot.ylabel('spike')
pyplot.legend()
pyplot.show()
```

(下页继续)

(续上页)

```
print('t', 'v', 'spike')
for t in range(T):
    print(t, v_list[t], spike_list[t])
```

运行后得到的电压和脉冲如下：



你会发现，LIF 神经元在有恒定输入电流时，电压会不断增大，但增速越来越慢。

如果输入电流不是足够大，最终在每个 dt 内，LIF 神经元的电压衰减值会恰好等于输入电流造成的电压增加值，电压不再增大，导致无法充电到过阈值、发放脉冲。

当停止输入后，LIF 神经元的电压会指数衰减，从图中 500 个 dt 后的曲线可以看出。

我们修改代码，给予更大的电流输入：

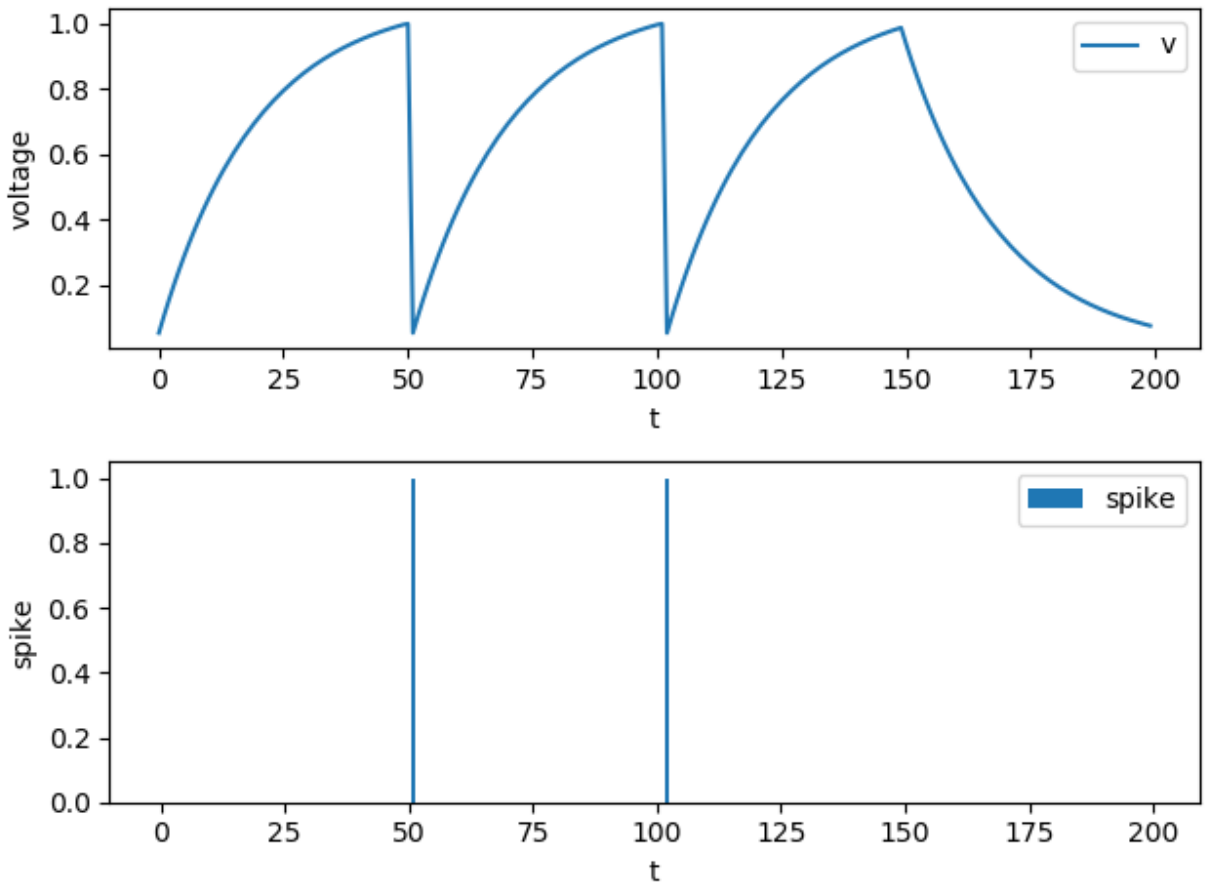
```
...
for t in range(T):
    # 前 150 次，输入电流都是 0.12
    if t < 150:
```

(下页继续)

(续上页)

```
        spike_list.append(lif_node(0.12).float().item())
...
```

运行后得到的电压和脉冲如下（需要说明的是，脉冲是以 pyplot 柱状图的形式画出，当柱状图的横轴，也就是时间太长时，而图像的宽度又不够大，一些“落单”的脉冲在图像上会无法画出，因为宽度小于一个像素点）：



可以发现，LIF 神经元已经开始发放脉冲了：

3.1.2.1.2 定义新的神经元

在 SNN 中，不同的神经元模型，区别往往体现在描述神经元的微分方程。上文所使用的 LIF 神经元，描述其动态特性的微分方程为：

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

其中 τ_m 是细胞膜的时间常数，$V(t)$ 是膜电位，V_{reset} 是静息电压，R_m 是膜电阻，$I(t)$ 是输入电流

SpikingFlow 是时间驱动（time-driven）的框架，即将微分方程视为差分方程，通过逐步仿真来进行计算。例如 LIF 神经元，代码位于 `SpikingFlow.neuron.LIFNode`，参考它的实现：

```
def forward(self, i):
    '''
    :param i: 当前时刻的输入电流，可以是一个 float，也可以是 tensor
    :return: out_spike: shape 与 self.shape 相同，输出脉冲
    '''
    out_spike = self.next_out_spike

    # 将上一个 dt 内过阈值的神经元重置
    if isinstance(self.v_reset, torch.Tensor):
        self.v[out_spike] = self.v_reset[out_spike]
    else:
        self.v[out_spike] = self.v_reset

    v_decay = -(self.v - self.v_reset)
    self.v += (self.r * i + v_decay) / self.tau
    self.next_out_spike = (self.v >= self.v_threshold)
    self.v[self.next_out_spike] = self.v_threshold
    self.v[self.v < self.v_reset] = self.v_reset

    return out_spike
```

从代码中可以发现，t-dt 时刻电压没有达到阈值，t 时刻电压达到了阈值，则到 t+dt 时刻才会放出脉冲。这是为了方便查看波形图，如果不这样设计，若 t-dt 时刻电压为 0.1，v_threshold=1.0，v_reset=0.0, t 时刻增加了 0.9，直接在 t 时刻发放脉冲，则从波形图上看，电压从 0.1 直接跳变到了 0.0，不利于进行数据分析。

此外，“脉冲”被定义为“torch.bool”类型的变量。SNN 中的神经元，输出的应该是脉冲而不是电压之类的其他值。

如果想自行实现其他类型的神经元，只需要继承 `SpikingFlow.neuron.BaseNode`，并实现 `__init__()`，`forward()`，`reset()` 函数即可。

3.1.2.2 编码器 SpikingFlow.encoding

本教程作者：fangwei123456

本节教程主要关注 `SpikingFlow.encoding`，包括如何使用已有编码器、如何定义新的编码器。

3.1.2.2.1 泊松编码器

SNN 中，脉冲在层与层之间传递信息。在 SpikingFlow 中，脉冲被定义为 `torch.bool` 类型的 tensor，也就是离散的 0 和 1。因此，常见的数据类型，例如图像，并不能直接输入到 SNN 中，需要做一些转换，将这些数据转换成脉冲，这一过程即为“编码”。

泊松编码器是最简单，但也效果良好、广泛使用的一种编码。

对于输入 $x \in [0, 1]$ ，泊松编码器将其编码为发放次数的分布符合泊松过程的脉冲。实现泊松过程非常简单，因为泊松流具有独立增量性、增量平稳性的性质。在一个仿真步长内，令发放脉冲的概率为 $p = x$ ，就可以实现泊松编码。

参考 `SpikingFlow.encoding.PoissonEncoder` 的代码：

```
def forward(self, x):
    '''
    :param x: 要编码的数据，任意形状的 tensor，要求 x 的数据范围必须在 [0, 1]

    将输入数据 x 编码为脉冲，脉冲发放的概率即为对应位置元素的值
    '''
    out_spike = torch.rand_like(x).le(x)
    # torch.rand_like(x) 生成与 x 相同 shape 的介于 [0, 1) 之间的随机数， 这个随机数小于等于 x
中对应位置的元素，则发放脉冲
    return out_spike
```

使用起来也比较简单：

```
pe = encoding.PoissonEncoder()
x = torch.rand(size=[8])
print(x)
for i in range(10):
    print(pe(x))
```

3.1.2.2.2 更复杂的编码器

更复杂的一些编码器，并不能像泊松编码器这样使用。例如某个编码器，将输入 0.1 编码成 `[0, 0, 1, 0]` 这样的脉冲，由于我们的框架是时间驱动的，因此编码器需要逐步输出 0, 0, 1, 0。

对于这样的编码器，例如 `SpikingFlow.encoding.LatencyEncoder`，调用时需要先编码一次，也就是条用 `forward()` 函数编码数据然后再输出 T 次，也就是调用 T 次 `step()`，例如：

```
x = torch.rand(size=[3, 2])
max_spike_time = 20
le = encoding.LatencyEncoder(max_spike_time)
```

(下页继续)

(续上页)

```
le(x)
print(x)
print(le.spike_time)
for i in range(max_spike_time):
    print(le.step())
```

3.1.2.2.3 定义新的编码器

编码器的实现非常灵活，因此在编码器的基类 `SpikingFlow.encoding.BaseEncoder` 中，并没有很多的限制。对于在一个仿真步长内就能输出所有脉冲的编码器，只需要实现 `forward()` 函数；对于需要多个仿真步长才能输出所有编码脉冲的编码器，需要实现 `forward()`,`step()`,`reset()` 函数。

3.1.2.3 仿真器 SpikingFlow.simulating

本教程作者：Yanqi-Chen

本节教程主要关注 `SpikingFlow.simulating`，包括如何使用仿真器。

3.1.2.3.1 仿真原理

所谓“仿真器”，即为将多个模块囊括其中，并统一运行的工具。实现 STDP 等学习功能，需要在仿真器层面进行操作。因此，如果想要实现其他类型的学习功能，需要对仿真器的实现过程有一定的了解。

采取时间驱动（time-driven）的模型均是由若干个 module 顺序连接而成，前一个 module 的输出作为后一个的输入。设输入为 x_0, 第 i 个 module M_i 的输出为 x_{i+1} ，仿真的数据流可以简单地描述为

$$x_0 \xrightarrow{M_0} x_1 \xrightarrow{M_1} \cdots \xrightarrow{M_{n-1}} x_n$$

仿真器建立时，首先将所有需要用到的 module 按顺序存放在 `module_list` 中，在每一个仿真时间步内，`x[i]` 将经由上述路径到达输出 `x[i+1]` 。仿真器用一个列表 `pipeline` 依次记录各个 module 在当前仿真时间步的输出，特别地，`pipeline[0]` 为输入。

在整个仿真过程中，输入保持固定，随着仿真时间的推移，脉冲逐渐向后面的 module 传递，如下面的代码所示。

```
...
for i in range(self.module_list.__len__(), 0, -1):
    #  x[n] = module[n-1](x[n-1])
    #  x[n-1] = module[n-2](x[n-2])
    #  ...
    #  x[1] = module[0](x[0])
    if self.pipeline[i - 1] is not None:
```

(下页继续)

(续上页)

```
        self.pipeline[i] = self.module_list[i - 1](self.pipeline[i - 1])
    else:
        self.pipeline[i] = None
...
```

对于输入尚未给出（神经信号还未传到）的 module，其输出定为 `None`。

可以看出，这里给出的仿真器中，每个神经信号在一个仿真步长内只能转移到连接的下一个模块。因此对于一个包含 n 个模块的模型，需要 n 个仿真时间步之后才会在最后一层有输出。

3.1.2.3.2 快速仿真

然而，很多时候我们希望模型在仿真的第一时间就给出输出，而不是慢慢等信号依次传递到最后一个模块才输出结果。为此，我们增加了一个快速仿真选项。仿真器初始化时 **默认开启**快速仿真，可以手动设置关闭。

若开启快速仿真，则在仿真器首次运行时，会运行 n 步而不是 1 步，并认为这 n 步的输入都是 input_data

```
...
# 快速仿真开启时，首次运行时跑满 pipeline
# x[0] -> module[0] -> x[1]
# x[1] -> module[1] -> x[2]
# ...
# x[n-1] -> module[n-1] -> x[n]
if self.simulated_steps == 0 and self.fast:
    for i in range(self.module_list.__len__()):
        # i = 0, 1, ..., n-1
        self.pipeline[i + 1] = self.module_list[i](self.pipeline[i])
...
```

3.1.2.4 突触连接 SpikingFlow.connection

本教程作者：fangwei123456

本节教程主要关注 `SpikingFlow.connection`，包括如何使用已有突触连接、如何定义新的突触连接。

3.1.2.4.1 脉冲电流转换器

SpikingFlow 中，神经元的输出都是 `torch.bool` 类型的脉冲，而输入则为 `torch.float` 类型的电流。将脉冲转换为电流的转换器，功能上类似于突触连接，但又有所不同，因此被视为突触连接包的子模块，定义在 `SpikingFlow.connection.transform` 中。

最简单的将脉冲转换为电流的方式，自然是不做任何处理，直接转换。`SpikingFlow.connection.transform.SpikeCurrent` 正是这样做的：

```
def forward(self, in_spike):
    '''
    :param in_spike: 输入脉冲
    :return: 输出电流

    简单地将输入脉冲转换为 0/1 的浮点数，然后乘以 amplitude
    '''
    return in_spike.float() * self.amplitude
```

其中 `self.amplitude` 是初始化时给定的放大系数。

更复杂的转换器，例如 `SpikingFlow.connection.transform.ExpDecayCurrent`，具有指数衰减的特性。若当前时刻到达一个脉冲，则电流变为 `amplitude`，否则电流按时间常数为 `tau` 进行指数衰减：

```
def forward(self, in_spike):
    '''
    :param in_spike: 输入脉冲
    :return: 输出电流
    '''
    in_spike_float = in_spike.float()
    i_decay = -self.i / self.tau
    self.i += i_decay * (1 - in_spike_float) + self.amplitude * in_spike_float
    return self.i
```

`ExpDecayCurrent` 可以看作是一个能够瞬间充满电的电容器，有脉冲作为输入时，则直接充满电；无输入时则自行放电。这种特性，使得 `ExpDecayCurrent` 与 `SpikeCurrent` 相比，具有了“记忆”，因而它需要额外定义一个重置状态的函数：

```
def reset(self):
    '''
    :return: None
    重置所有状态变量为初始值，对于 ExpDecayCurrent 而言，直接将电流设置为 0 即可
    '''
    self.i = 0
```

3.1.2.4.2 定义新的脉冲电流转换器

从之前的例子也可以看出，脉冲电流转换器的定义比较简单，接受 `torch.bool` 类型的脉冲作为输入，输出 `torch.float` 类型的电流。只需要继承 `SpikingFlow.connection.transform.BaseTransformer`，实现 `__init__()` 和 `forward()` 函数，对于有记忆（状态）的转换器，则额外实现一个 `reset()` 函数。

3.1.2.4.3 突触连接

SpikingFlow 里的突触连接，与传统 ANN 中的连接非常类似，都可以看作是一个简单的矩阵，而信息在突触上的传递，则可以看作是矩阵运算。

例如 `SpikingFlow.connection.Linear`，与 PyTorch 中的 `nn.Linear` 的行为基本相同：

```
def forward(self, x):
    '''
    :param x: 输入电流，shape=[batch_size, *, in_num]
    :return: 输出电流，shape=[batch_size, *, out_num]
    '''
    return torch.matmul(x, self.w.t())
```

3.1.2.4.4 定义新的突触连接

定义新的突触连接，与定义新的脉冲电流转换器非常类似，只需要继承 `SpikingFlow.connection.BaseConnection`，实现 `__init__()` 和 `forward()` 函数。对于有记忆（状态）的突触，也需要额外实现 `reset()` 函数。

3.1.2.5 学习规则 SpikingFlow.learning

本教程作者：fangwei123456

本节教程主要关注 `SpikingFlow.learning`，包括如何使用已有学习规则、如何定义新的学习规则。

3.1.2.5.1 学习规则是什么

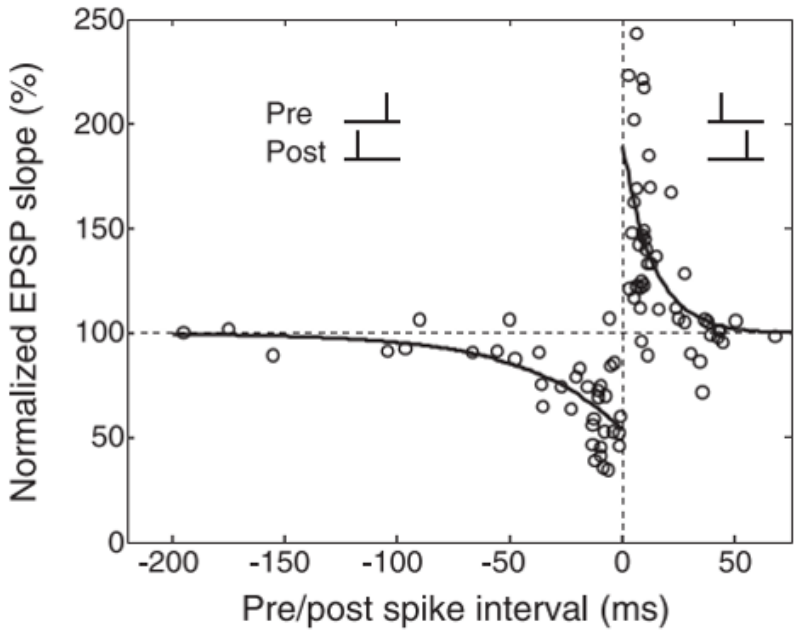
“学习”在 ANN 中或许更多地被称作是“训练”。ANN 中基于梯度的反向传播优化算法，就是应用最为广泛的学习规则。

在 SNN 中，发放脉冲这一过程通常使用阶跃函数去描述，这是一个不可微分的过程；SNN 比较注重生物可解释性，生物神经系统中似乎并没有使用反向传播这种训练成千上万次才能达到较好结果的“低效率”方法。在 SNN 中如何使用反向传播算法也是一个研究热点。使用反向传播算法的 SNN 一般为事件驱动模型（例如 SpikeProp 和 Tempotron，在 `SpikingFlow.event_driven` 中可以找到），但也有一些算法会使用时间驱动模型，可见于 `SpikingFlow.softbp`。`SpikingFlow.learning` 中更多的聚焦于生物可解释性的学习算法，例如 STDP。

3.1.2.5.2 STDP(Spike Timing Dependent Plasticity)

STDP(Spike Timing Dependent Plasticity) 学习规则是在生物实验中发现的一种突触可塑性机制。实验发现，突触的连接强度受到突触连接的前（pre）后（post）神经元脉冲活动的影响。

如果 pre 神经元先发放脉冲，post 神经元后发放脉冲，则突触强度增大；反之，如果 post 神经元先发放脉冲，pre 神经元后发放脉冲，则突触强度减小。生物实验数据如下图所示，横轴是 pre 神经元和 post 神经元释放的一对脉冲的时间差，也就是 $t_{post} - t_{pre}$，纵轴表示突触强度变化的百分比：



这种突触强度和前后脉冲发放时间的关系，可以用以下公式进行拟合：

$$\Delta w = \begin{cases} Ae^{\frac{t_{pre}-t_{post}}{\tau}}, t_{pre} - t_{post} \leq 0, A > 0 \\ Be^{-\frac{t_{pre}-t_{post}}{\tau}}, t_{pre} - t_{post} \geq 0, B < 0 \end{cases} \tag{3.1}$$

一般认为，突触连接权重的改变，是在脉冲发放的瞬间完成。不过，上图中的公式并不适合代码去实现，因为它需要分别记录前后神经元的脉冲发放时间。使用[1] 提供的基于双脉冲的迹的方式来实现 STDP 更为优雅。

对于突触的 pre 神经元 j 后 post 神经元 i，分别使用一个名为迹（trace）的变量 x_j, y_i，迹由类似于 LIF 神经元的膜电位的微分方程来描述：

$$\frac{\mathrm{d}x_j}{\mathrm{d}t} = -\frac{x_j}{\tau_x} + \sum_{t_j^f} \delta(t - t_j^f)$$

$$\frac{\mathrm{d}y_i}{\mathrm{d}t} = -\frac{y_i}{\tau_y} + \sum_{t_i^f} \delta(t - t_i^f)$$

其中 t_j^f, t_i^f 是 pre 神经元 j 后 post 神经元 i 的脉冲发放时刻，$\delta(t)$ 是脉冲函数，只在 $t = 0$ 处为 1，其他时刻均为 0。

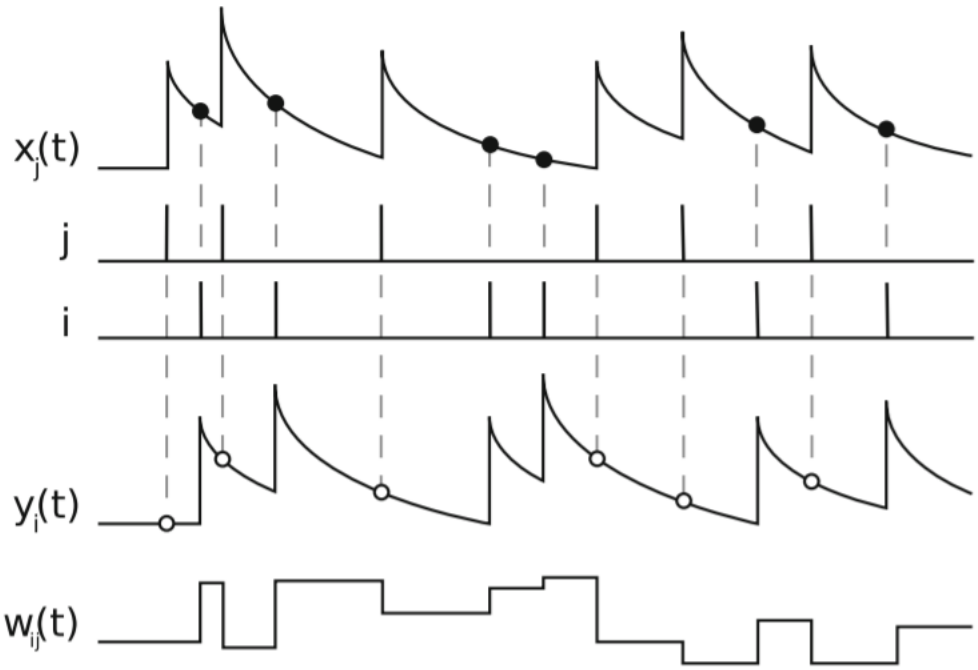
[1] Morrison A, Diesmann M, Gerstner W. Phenomenological models of synaptic plasticity based on spiketiming[J]. Biological cybernetics, 2008, 98(6): 459-478.

当 pre 神经元 j 的脉冲 t_j^f 到达时，突触权重减少；当 post 神经元 i 的脉冲 t_i^f 到达时，突触权重增加：

$$\Delta w_{ij}^-(t_j^f) = -F_-(w_{ij})y_i(t_j^f)$$
$$\Delta w_{ij}^+(t_i^f) = -F_+(w_{ij})x_j(t_i^f)$$

其中 $F_+(w_{ij}), F_-(w_{ij})$ 是突触权重 w_{ij} 的函数，控制权重的增量。

[1] 中给出了这种方式的示意图：



`SpikingFlow.learning.STDPModule` 是使用迹的方式实现的一个 STDP 学习模块。`STDPModule` 会将脉冲电流转换器 `tf_module`、突触 `connection_module`、神经元 `neuron_module` 三者打包成一个模块，将输入到 `tf_module` 的脉冲，作为 pre 神经元的脉冲；`neuron_module` 输出的脉冲，作为 post 神经元的脉冲，利用 STDP 学习规则，来更新 `connection_module` 的权重。

示例代码如下：

```
import SpikingFlow.simulating as simulating
import SpikingFlow.learning as learning
import SpikingFlow.connection as connection
import SpikingFlow.connection.transform as tf
import SpikingFlow.neuron as neuron
import torch
from matplotlib import pyplot

# 新建一个仿真器
sim = simulating.Simulator()

# 添加各个模块。为了更明显的观察到脉冲，我们使用 IF 神经元，而且把膜电阻设置的很大
```

(下页继续)

(续上页)

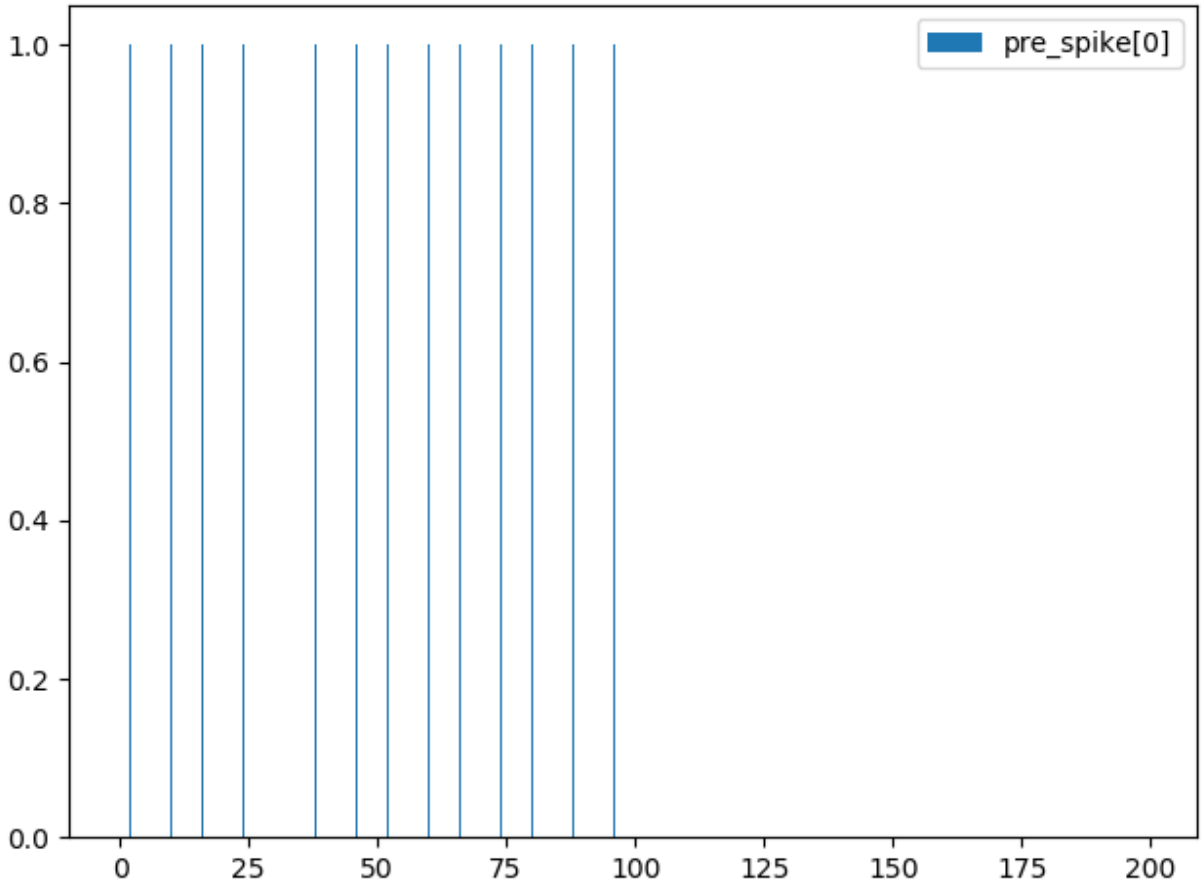
```
# 突触的 pre 是 2 个输入，而 post 是 1 个输出，连接权重是 shape=[1, 2] 的 tensor
sim.append(learning.STDPModule(tf.SpikeCurrent(amplitude=0.5),
                               connection.Linear(2, 1),
                               neuron.IFNode(shape=[1], r=50.0, v_threshold=1.0),
                               tau_pre=10.0,
                               tau_post=10.0,
                               learning_rate=1e-3
                               ))
# 新建 list，分别保存 pre 的 2 个输入脉冲、post 的 1 个输出脉冲，以及对应的连接权重
pre_spike_list0 = []
pre_spike_list1 = []
post_spike_list = []
w_list0 = []
w_list1 = []
T = 200

for t in range(T):
    if t < 100:
        # 前 100 步仿真，pre_spike[0] 和 pre_spike[1] 都是发放一次 1 再发放一次 0
        if t % 2 == 0:
            pre_spike = torch.ones(size=[2], dtype=torch.bool)
        else:
            pre_spike = torch.zeros(size=[2], dtype=torch.bool)
    else:
        # 后 100 步仿真，pre_spike[0] 一直为 0，而 pre_spike[1] 一直为 1
        pre_spike = torch.zeros(size=[2], dtype=torch.bool)
        pre_spike[1] = True

    post_spike = sim.step(pre_spike)
    pre_spike_list0.append(pre_spike[0].float().item())
    pre_spike_list1.append(pre_spike[1].float().item())

    post_spike_list.append(post_spike.float().item())

    w_list0.append(sim.module_list[-1].module_list[2].w[:, 0].item())
    w_list1.append(sim.module_list[-1].module_list[2].w[:, 1].item())

# 画出 pre_spike[0]
pyplot.bar(torch.arange(0, T).tolist(), pre_spike_list0, width=0.1, label='pre_
↪spike[0]')
pyplot.legend()
pyplot.show()
```

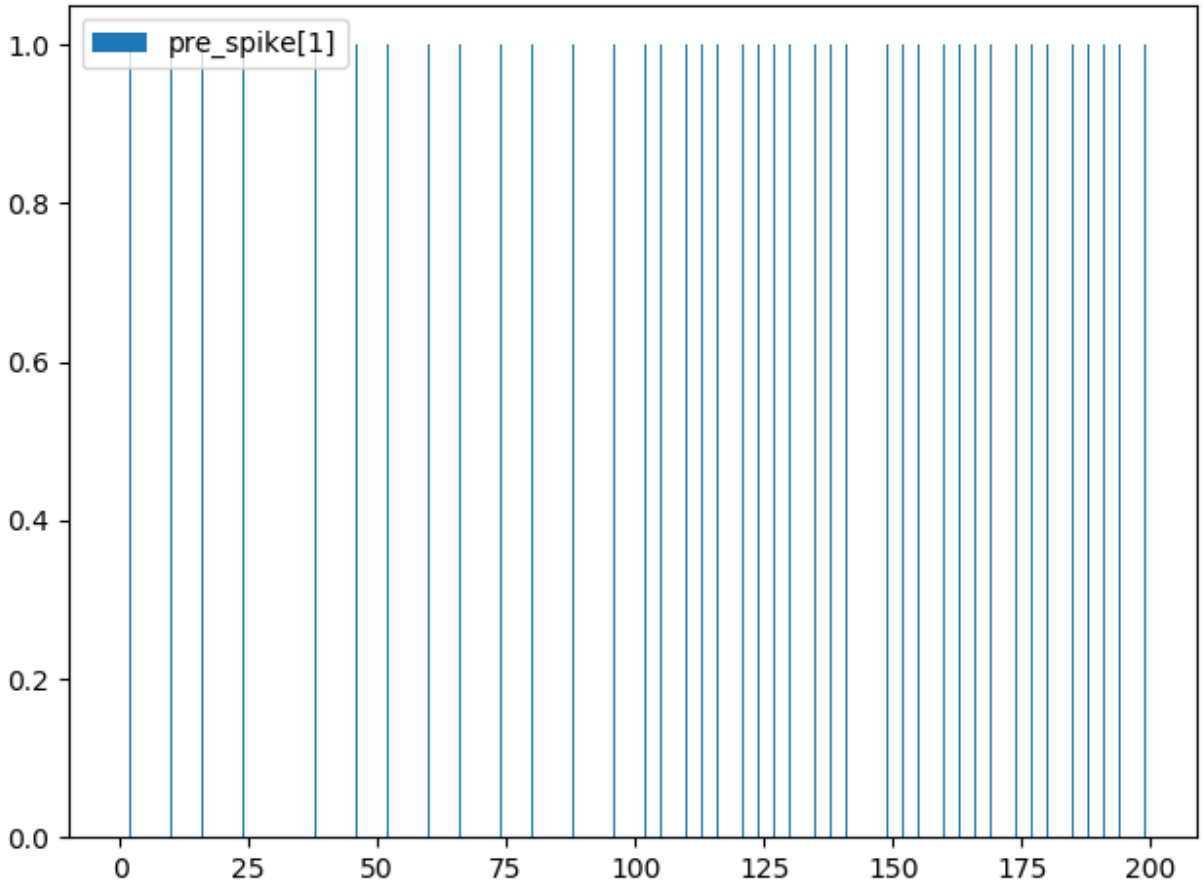
(下页继续)

(续上页)

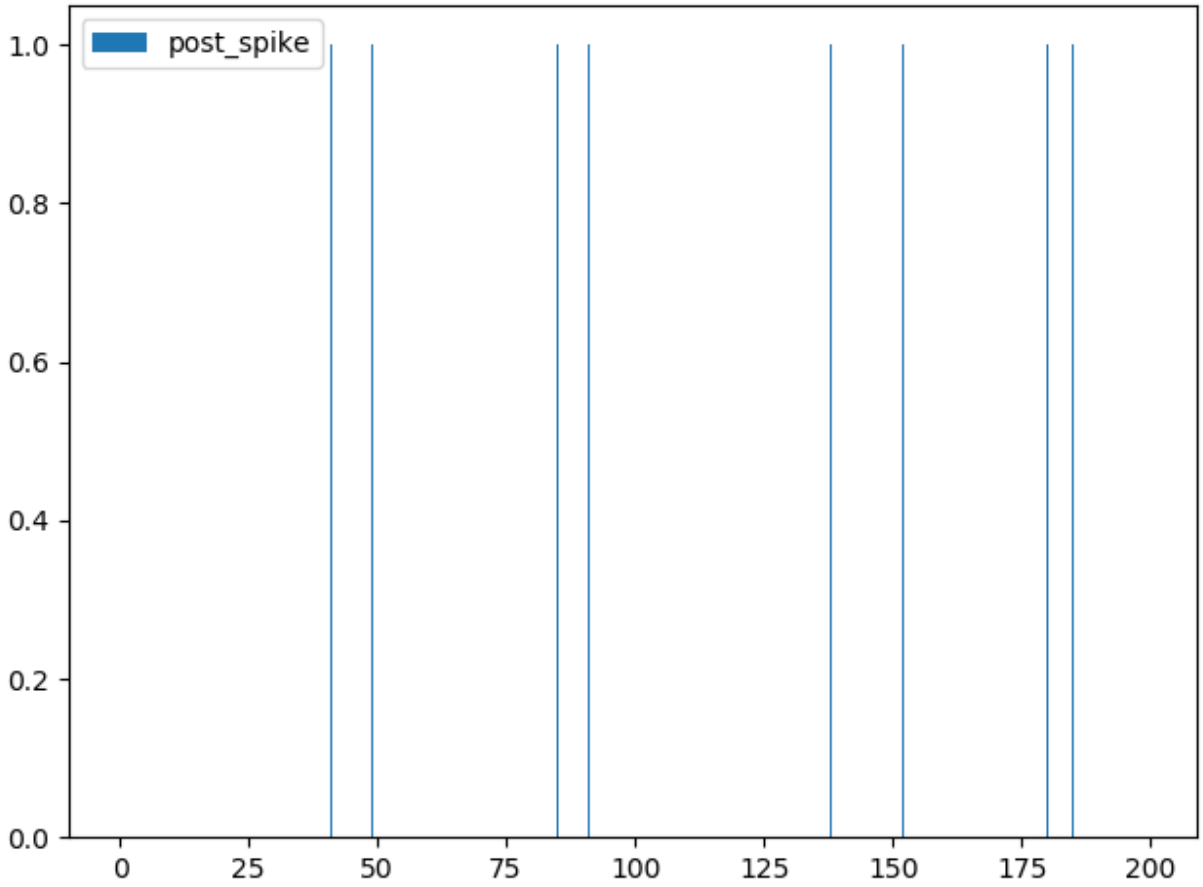
```
# 画出 pre_spike[1]
pyplot.bar(torch.arange(0, T).tolist(), pre_spike_list1, width=0.1, label='pre_
↪spike[1]')
pyplot.legend()
pyplot.show()

# 画出 post_spike
pyplot.bar(torch.arange(0, T).tolist(), post_spike_list, width=0.1, label='post_spike
↪')
pyplot.legend()
pyplot.show()

# 画出 2 个输入与 1 个输出的连接权重 w_0 和 w_1
pyplot.plot(w_list0, c='r', label='w[0]')
pyplot.plot(w_list1, c='g', label='w[1]')
pyplot.legend()
pyplot.show()
```

这段代码中，突触的输入是 2 个脉冲，而输出是 1 个脉冲，在前 100 步仿真中，`pre_spike[0]` 和 `pre_spike[1]` 都每隔 1 个仿真步长发放 1 次脉冲，而在后 100 步仿真，`pre_spike[0]` 停止发放，`pre_spike[1]` 持续发放，如下图所示（需要说明的是，脉冲是以 pyplot 柱状图的形式画出，当柱状图的横轴，也就是时间太长时，而图像的宽度又不够大，一些“落单”的脉冲在图像上会无法画出，因为宽度小于一个像素点）：

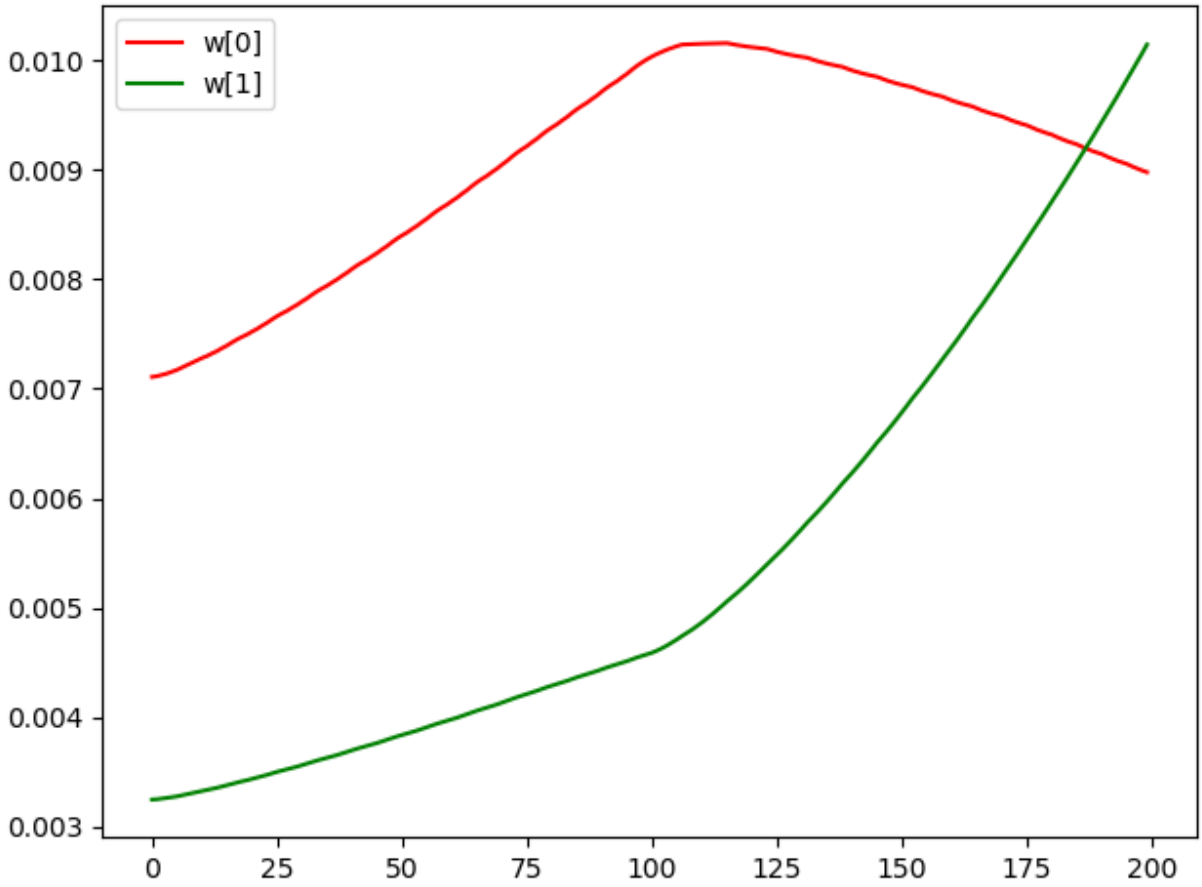

pre_spike[0]
1.0
0.8
0.6
0.4
0.2
0.0
0
25
50
75
100
125
150
175
200



引发的 post 神经元的脉冲如下图：



在前 100 步，w_{00}, w_{01} 均增大；而后 100 步，由于我们人为设定 `pre_spike[0]` 停止发放，`pre_spike[1]` 持续发放，故 w_{00} 减小，w_{01} 增大：



3.1.2.5.3 更灵活的 STDPUpdater

在 `SpikingFlow.learning.STDPModule` 中将脉冲电流转换器、突触、神经元这 3 个模块封装为 1 个，简化了使用，但封装也带来了灵活性的缺失。`SpikingFlow.learning.STDPUpdater` 则提供了一种更为灵活的使用方式，可以手动地设置突触和其对应的前后脉冲，即便“前后脉冲”并不是真正的突触连接的前后神经元的脉冲，也可以被用来“远程更新”突触的权重。

示例代码如下，与 `STDPModule` 的示例类似：

```
import SpikingFlow.simulating as simulating
import SpikingFlow.learning as learning
import SpikingFlow.connection as connection
import SpikingFlow.connection.transform as tf
import SpikingFlow.neuron as neuron
import torch
from matplotlib import pyplot

# 定义权值函数 f_w
```

(下页继续)

(续上页)

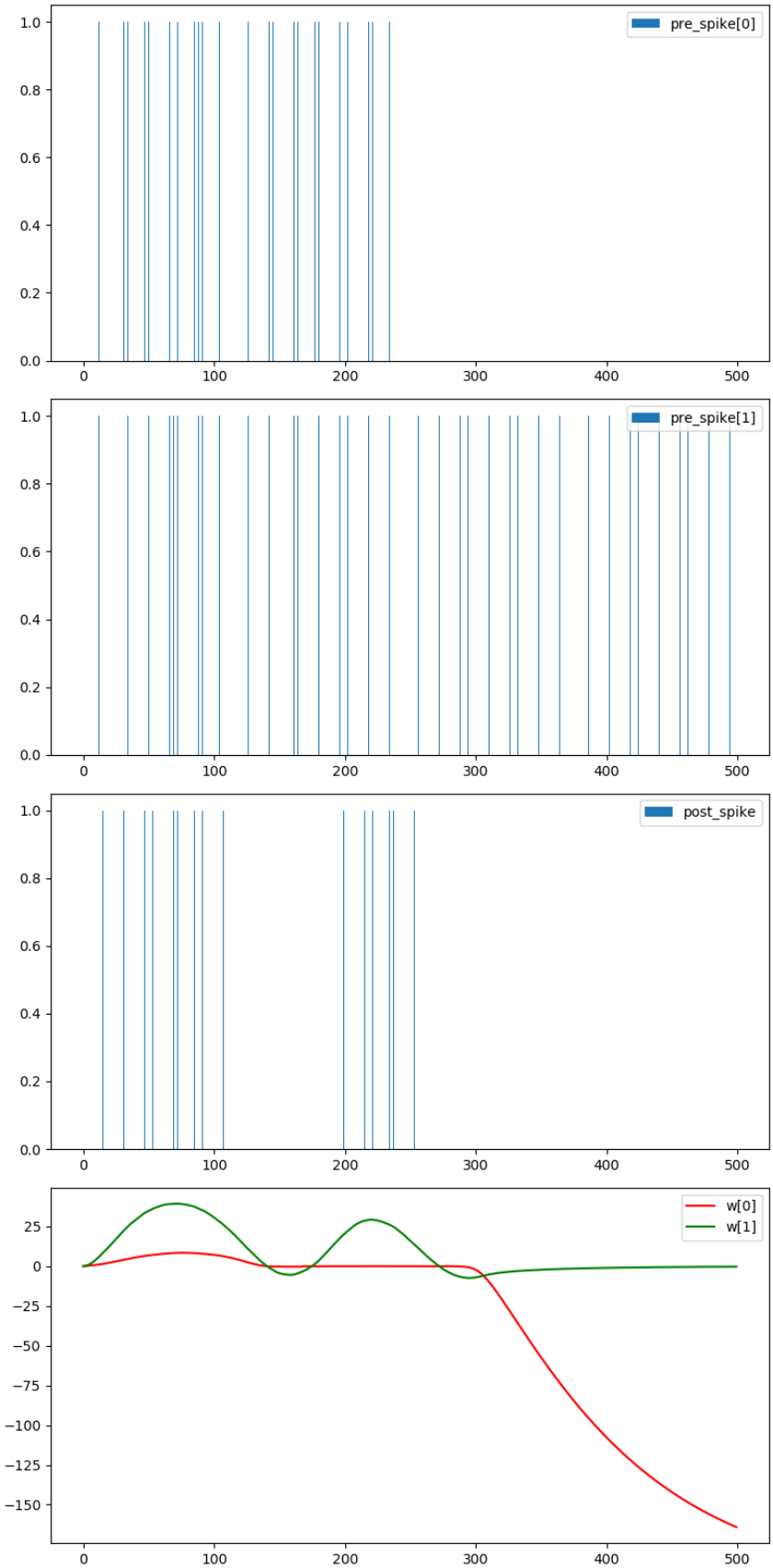
```
def f_w(x: torch.Tensor):
    x_abs = x.abs()
    return x_abs / (x_abs.sum() + 1e-6)

# 新建一个仿真器
sim = simulating.Simulator()

# 放入脉冲电流转换器、突触、LIF 神经元
sim.append(tf.SpikeCurrent(amplitude=0.5))
sim.append(connection.Linear(2, 1))
sim.append(neuron.LIFNode(shape=[1], r=10.0, v_threshold=1.0, tau=100.0))

# 新建一个 STDPUpdater
updater = learning.STDPUpdater(tau_pre=50.0,
                               tau_post=100.0,
                               learning_rate=1e-1,
                               f_w=f_w)

# 新建 list, 保存 pre 脉冲、post 脉冲、突触权重 w_00, w_01
pre_spike_list0 = []
pre_spike_list1 = []
post_spike_list = []
w_list0 = []
w_list1 = []

T = 500
for t in range(T):
    if t < 250:
        if t % 2 == 0:
            pre_spike = torch.ones(size=[2], dtype=torch.bool)
        else:
            pre_spike = torch.randint(low=0, high=2, size=[2]).bool()
    else:
        pre_spike = torch.zeros(size=[2], dtype=torch.bool)
        if t % 2 == 0:
            pre_spike[1] = True




    pre_spike_list0.append(pre_spike[0].float().item())
    pre_spike_list1.append(pre_spike[1].float().item())
```

(下页继续)

(续上页)

```
    post_spike = sim.step(pre_spike)

    updater.update(sim.module_list[1], pre_spike, post_spike)

    post_spike_list.append(post_spike.float().item())

    w_list0.append(sim.module_list[1].w[:, 0].item())
    w_list1.append(sim.module_list[1].w[:, 1].item())

pyplot.figure(figsize=(8, 16))
pyplot.subplot(4, 1, 1)
pyplot.bar(torch.arange(0, T).tolist(), pre_spike_list0, width=0.1, label='pre_
↪spike[0]')
pyplot.legend()

pyplot.subplot(4, 1, 2)
pyplot.bar(torch.arange(0, T).tolist(), pre_spike_list1, width=0.1, label='pre_
↪spike[1]')
pyplot.legend()

pyplot.subplot(4, 1, 3)
pyplot.bar(torch.arange(0, T).tolist(), post_spike_list, width=0.1, label='post_spike
↪')
pyplot.legend()

pyplot.subplot(4, 1, 4)
pyplot.plot(w_list0, c='r', label='w[0]')
pyplot.plot(w_list1, c='g', label='w[1]')
pyplot.legend()
pyplot.show()
```

运行结果如下：


pre_spike[0]
pre_spike[1]
post_spike
w[0]
w[1]

3.1.2.5.4 定义新的学习规则

定义新的学习规则，可以参考 `STDPModule` 和 `STDPUpdater` 的代码。需要注意的是，对于每一种突触类型，都应该实现一个对应的参数更新方式，例如 `STDPUpdater` 的如下代码：

```
def update(self, connection_module, pre_spike, post_spike, inverse=False):
...
if isinstance(connection_module, connection.Linear):
    ...
...
```

上述代码是针对 `SpikingFlow.connection.Linear` 进行的特定实现。

3.1.2.6 软反向传播 SpikingFlow.softbp

本教程作者：fangwei123456

本节教程主要关注 `SpikingFlow.softbp`，介绍软反向传播的概念、可微分 SNN 神经元的使用方式。

需要注意的是，`SpikingFlow.softbp` 是一个相对独立的包，与其他的 `SpikingFlow.*` 中的神经元、突触等组件不能混用。

软反向传播的灵感，来源于以下两篇文章:

Mentzer F, Agustsson E, Tschannen M, et al. Conditional probability models for deep image compression[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018: 4394-4402.

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.

3.1.2.6.1 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，但为了方便，在 `SpikingFlow.softbp` 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。

可以用以下描述方程来描述任意的 SNN：

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

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电压；$f(V(t-1), X(t))$ 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，状态更新方程，及其离散化的方程如下：

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

由于状态更新方程不能描述脉冲发放的过程，因此我们用 $H(t)$ 来代替 $V(t)$，用 $V(t)$ 表示完成脉冲发放（或者不发放）过程后的神经元膜电压。

$S(t)$ 是神经元发放的脉冲，$g(x) = \Theta(x)$ 是阶跃函数，或者按 RNN 的习惯称为门控函数，输出仅为 0 或 1，可以表示脉冲的发放过程，定义为

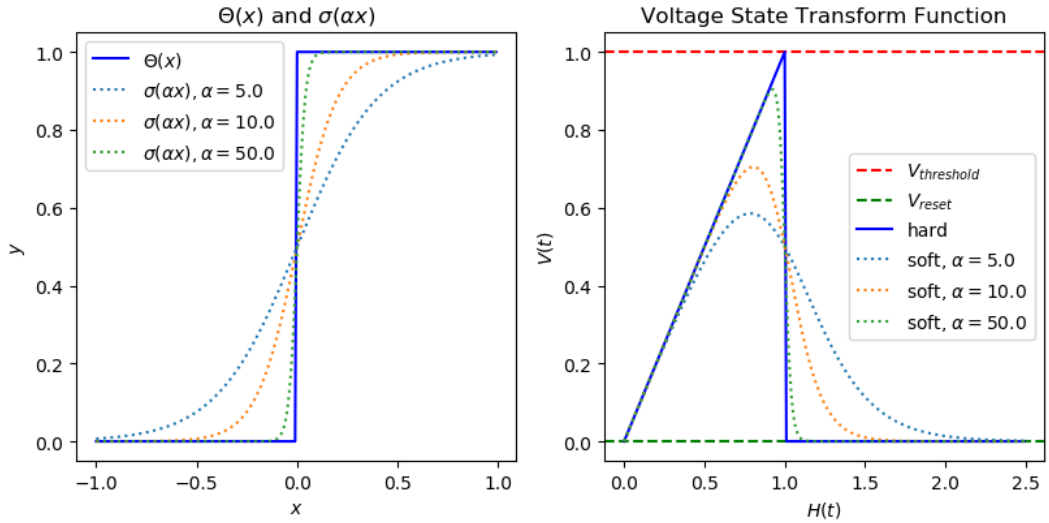
$$\Theta(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

发放脉冲，则电压重置为 V_{reset}；没有发放脉冲，则电压不变，这就是描述方程的最后一个方程，电压状态转移方程。

3.1.2.6.2 硬前向与软反向

RNN 使用可微分的门控函数，例如 tanh 函数。而 SNN 的门控函数 $g(x) = \Theta(x)$ 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。

我们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，使用近似门控函数 $g'(x) = \sigma'(x)$ 来求梯度。最常见的近似 $g(x) = \Theta(x)$ 的门控函数即为 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 $\Theta(x)$ 但梯度越不光滑，网络也会越难以训练。近似门控函数引入后，电压状态转移函数 $V(t) = H(t) \cdot (1 - S(t)) + V_{reset} \cdot S(t)$ 也会随之改变。下图显示了不同的 α 以及电压状态转移方程：



默认的近似门控函数为 `SpikingFlow.softbp.soft_pulse_function.Sigmoid()`。近似门控函数是 `softbp` 包中基类神经元构造函数的参数之一：

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

在 SpikingFlow.softbp.soft_pulse_function 中还提供了其他的可选近似门控函数。

如果想要自定义新的近似门控函数，可以参考 soft_pulse_function.Sigmoid() 的代码实现：

```
class sigmoid(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, alpha):
        if x.requires_grad:
            alpha_x = x * alpha
            ctx.save_for_backward(alpha_x)
            ctx.alpha = alpha
        return (alpha_x >= 0).float()

    @staticmethod
    def backward(ctx, grad_output):
        grad_x = None
        if ctx.needs_input_grad[0]:
            alpha_x = ctx.saved_tensors[0]
            s_x = torch.sigmoid(alpha_x)
            grad_x = grad_output * s_x * (1 - s_x) * ctx.alpha
        return grad_x, None
```

3.1.2.6.3 作为激活函数的 SNN 神经元

解决了 SNN 的微分问题后，我们的 SNN 神经元可以像激活函数那样，嵌入到使用 PyTorch 搭建的任意网络中去了。在 SpikingFlow.softbp.neuron 中已经实现了 IF 神经元和 LIF 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),
```

（下页继续）

(续上页)

```
        neuron.LIFNode(tau=100.0, v_threshold=1.0, v_reset=5.0)
        )
```

3.1.2.6.4 MNIST 分类

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

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import sys
sys.path.append('.')
import SpikingFlow.softbp.neuron as neuron
import SpikingFlow.encoding as encoding
from torch.utils.tensorboard import SummaryWriter
import readline

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

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

    def reset_(self):
        for item in self.modules():
            if hasattr(item, 'reset'):
                item.reset()
def main():
    device = input('输入运行的设备，例如 “CPU” 或 “cuda:0”   ')
    dataset_dir = input('输入保存 MNIST 数据集的位置，例如 “./”   ')
    batch_size = int(input('输入 batch_size，例如 “64”   '))
```

(下页继续)

(续上页)

```
    learning_rate = float(input('输入学习率，例如“1e-3”  '))
    T = int(input('输入仿真时长，例如“50”  '))
    tau = float(input('输入 LIF 神经元的时间常数 tau，例如“100.0”  '))
    train_epoch = int(input('输入训练轮数，即遍历训练集的次数，例如“100”  '))
    log_dir = input('输入保存 tensorboard 日志文件的位置，例如“./”  ')

    writer = SummaryWriter(log_dir)

    # 初始化数据加载器
    train_data_loader = torch.utils.data.DataLoader(
        dataset=torchvision.datasets.MNIST(
            root=dataset_dir,
            train=True,
            transform=torchvision.transforms.ToTensor(),
            download=True),
        batch_size=batch_size,
        shuffle=True,
        drop_last=True)
    test_data_loader = torch.utils.data.DataLoader(
        dataset=torchvision.datasets.MNIST(
            root=dataset_dir,
            train=False,
            transform=torchvision.transforms.ToTensor(),
            download=True),
        batch_size=batch_size,
        shuffle=True,
        drop_last=False)

    # 初始化网络
    net = Net(tau=tau).to(device)
    # 使用 Adam 优化器
    optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
    # 使用泊松编码器
    encoder = encoding.PoissonEncoder()
    train_times = 0
    for _ in range(train_epoch):
        net.train()
        for img, label in train_data_loader:
            img = img.to(device)
            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
```

(下页继续)

(续上页)

```
            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，与真实类别的交叉熵
            # 这样的损失函数会使，当类别 i 输入时，输出层中第 i 个神经元的脉冲发放频率趋近 1，而其他
神经元的脉冲发放频率趋近 0
            loss = F.cross_entropy(out_spikes_counter_frequency, label.to(device))
            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
            net.reset_()

            # 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
            correct_rate = (out_spikes_counter_frequency.max(1)[1] == label.
↪to(device)).float().mean().item()
            writer.add_scalar('train_correct_rate', correct_rate, train_times)
            if train_times % 1024 == 0:
                print(device, dataset_dir, batch_size, learning_rate, T, tau, train_
↪epoch, log_dir)
                print('train_times', train_times, 'train_correct_rate', correct_rate)
            train_times += 1

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

                correct_sum += (out_spikes_counter.max(1)[1] == label.to(device)).
↪float().sum().item()
```

(下页继续)

(续上页)

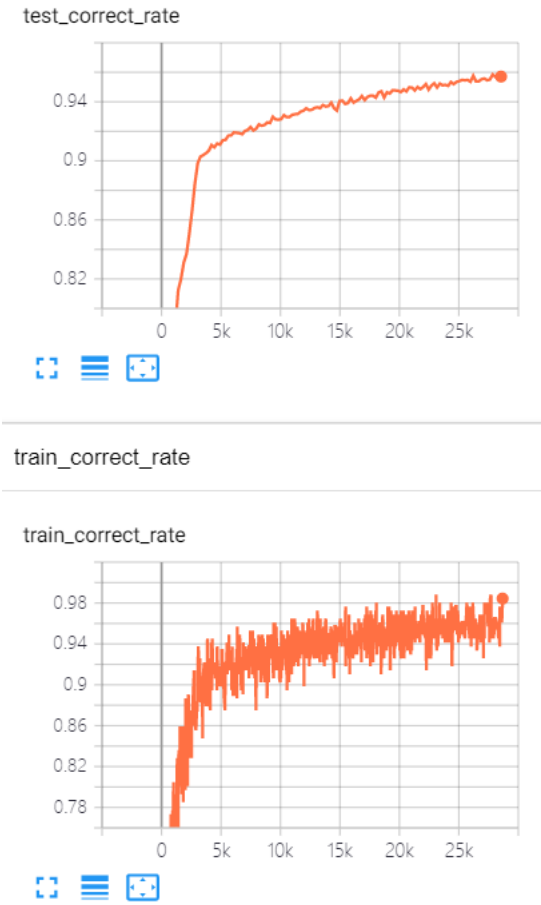
```
                test_sum += label.numel()
                net.reset_()

            writer.add_scalar('test_correct_rate', correct_sum / test_sum, train_
↪times)

if __name__ == '__main__':
    main()
```

这份代码位于 SpikingFlow.softbp.examples.mnist.py。进入 SpikingFlow 的根目录（也就是 GitHub 仓库的根目录），直接运行即可，例如：

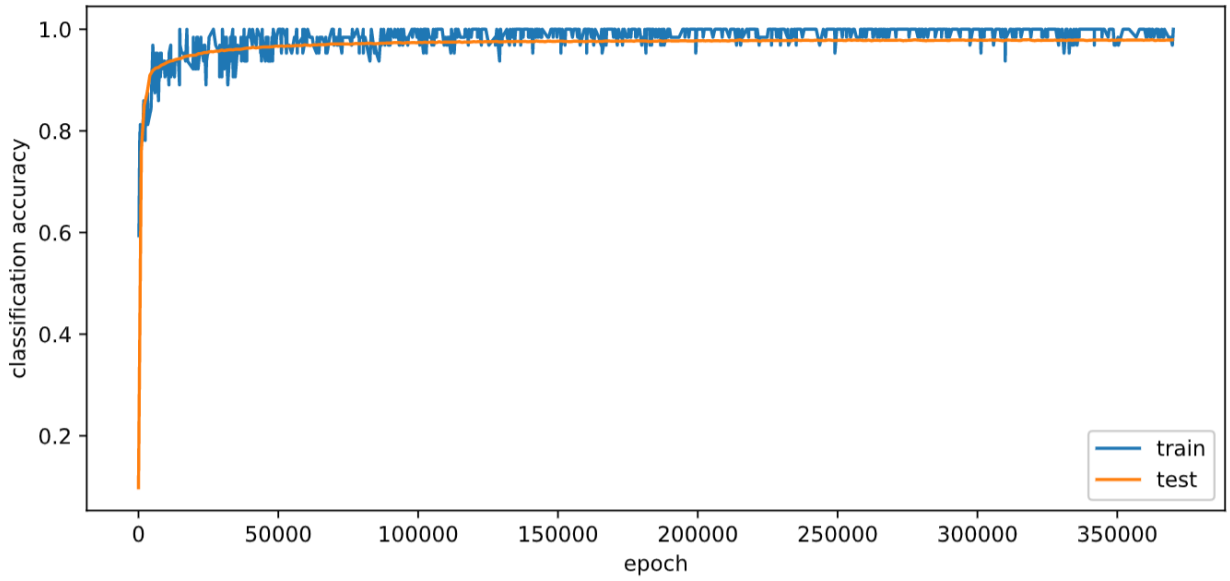
```
(pytorch-env) wfang@pami:~/SpikingFlow$ python SpikingFlow/softbp/examples/mnist.py
输入运行的设备，例如“CPU”或“cuda:0”  cuda:0
输入保存 MNIST 数据集的位置，例如“./”  ./tempdir
输入 batch_size，例如“64”  256
输入学习率，例如“1e-3”  1e-2
输入仿真时长，例如“50”  50
输入 LIF 神经元的时间常数 tau，例如“100.0”  100.0
输入训练轮数，即遍历训练集的次数，例如“100”  1000
输入保存 tensorboard 日志文件的位置，例如“./”  ./tempdir
```

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”，训练效果也一般更好。这个模型只占用了 276MB 显存，但在之后的 CIFAR10 示例中，由于 CNN 的引入，使得显存消耗量剧增。

我们的这个模型，在 Tesla K80 上训练一个半小时，tensorboard 记录的数据如下所示：



这个模型最终能够达到 98% 的测试集正确率，如下图所示，注意下图中的“epoch”表示训练次数，而代码中的“epoch”表示遍历一次训练集：



如果使用训练集增强的方法，例如给训练集图片加上一些随机噪声、仿射变换等，则训练好的网络泛化能力会进一步提升，最高能达到 99% 以上的测试集正确率。

3.1.2.6.5 CIFAR10 分类

我们的这种方法，具有的一大优势就是可以无缝嵌入到任意的 PyTorch 搭建的网络中。因此 CNN 的引入是非常简单而自然的。我们用 CNN 来进行 CIFAR10 分类任务，训练的代码与进行 MNIST 分类几乎相同，只需要更改一下网络结构和数据集。

```
class Net(nn.Module):
    def __init__(self, tau=100.0, v_threshold=1.0, v_reset=0.0):
        super().__init__()
        # 网络结构，卷积-卷积-最大池化堆叠，最后接一个全连接层
        self.conv = nn.Sequential(
            nn.Conv2d(3, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.MaxPool2d(2, 2),
            nn.BatchNorm2d(256),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset),  # 16␣
↪* 16

            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.MaxPool2d(2, 2),
            nn.BatchNorm2d(256),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset),  # 8 *␣
↪8

            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.MaxPool2d(2, 2),
            nn.BatchNorm2d(256),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset),  # 4 *␣
↪4

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

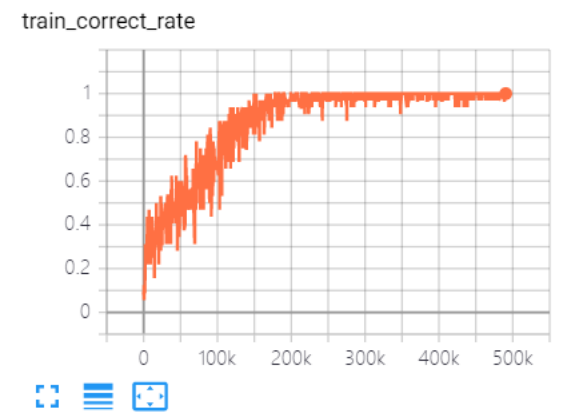
(下页继续)

(续上页)

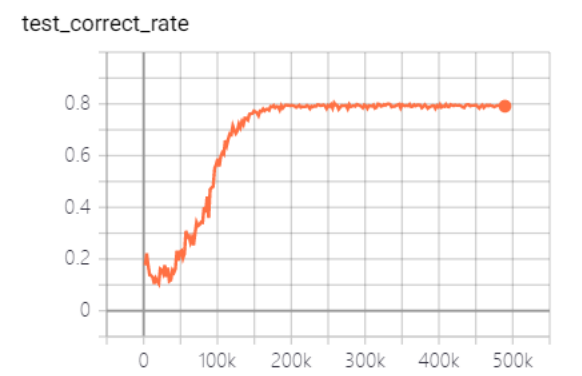
```
            nn.Linear(256 * 4 * 4, 10, bias=False),
            neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset)
                                )

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

    def reset_(self):
        for item in self.modules():
            if hasattr(item, 'reset'):
                item.reset()
def main():
    device = input('输入运行的设备，例如“CPU”或“cuda:0”   ')
    dataset_dir = input('输入保存 CIFAR10 数据集的位置，例如“./”   ')
    batch_size = int(input('输入 batch_size，例如“64”   '))
    learning_rate = float(input('输入学习率，例如“1e-3”   '))
    T = int(input('输入仿真时长，例如“50”   '))
    tau = float(input('输入 LIF 神经元的时间常数 tau，例如“100.0”   '))
    train_epoch = int(input('输入训练轮数，即遍历训练集的次数，例如“100”   '))
    log_dir = input('输入保存 tensorboard 日志文件的位置，例如“./”   ')

    writer = SummaryWriter(log_dir)

    # 初始化数据加载器
    train_data_loader = torch.utils.data.DataLoader(
        dataset=torchvision.datasets.CIFAR10(
            root=dataset_dir,
            train=True,
            transform=torchvision.transforms.ToTensor(),
            download=True),
        batch_size=batch_size,
        shuffle=True,
        drop_last=True)
    test_data_loader = torch.utils.data.DataLoader(
        dataset=torchvision.datasets.CIFAR10(
            root=dataset_dir,
            train=False,
            transform=torchvision.transforms.ToTensor(),
            download=True),
        batch_size=batch_size,
        shuffle=True,
        drop_last=False)
     # 后面的代码与 MNIST 分类相同，不再展示
```

这份代码位于 `SpikingFlow.softbp.examples.cifar10.py`，运行方法与之前的 MNIST 的代码相同。需要注意的是，由于 CNN 的引入，CNN 层后也跟有 LIF 神经元，CNN 层的输出是一个高维矩阵，因此其后的 LIF 神经元数量众多，导致这个模型极端消耗显存。在大约 `batch_size=32`，仿真时长 `T=50` 的情况下，这个模型几乎要消耗 `12G` 的显存。训练这样庞大模型，Tesla K80 的算力显得捉襟见肘。我们在 TITAN RTX 上训练大约 60 小时，网络才能收敛，测试集正确率大约为 80%。使用训练集增强的方法，同样可以提高泛化能力。

train_correct_rate



test_correct_rate



3.1.2.6.6 模型流水线

如前所述，在包含 SNN 神经元的网络中引入 CNN 后，显存的消耗量剧增。有时一个网络太大，以至于单个 GPU 无法放下。在这种情况下，我们可以将一个网络分割到多个 GPU 存放，充分利用多 GPU 闲置显存的优势。但使用这一方法，数据需要在多个 GPU 之间来回复制，在一定程度上会降低训练速度。

`SpikingFlow.softbp.ModelPipeline` 是一个基于流水线多 GPU 串行并行的基类，使用者只需要继承 `ModelPipeline`，然后调用 `append(nn_module, gpu_id)`，就可以将 `nn_module` 添加到流水线中，并且 `nn_module` 会被运行在 `gpu_id` 上。在调用模型进行计算时，`forward(x, split_sizes)` 中的 `split_sizes` 指的是输入数据 `x` 会在维度 0 上被拆分成每 `split_size` 一组，得到 `[x[0], x[1], ...]`，这些数据会被串行的送入 `module_list` 中保存的各个模块进行计算。

例如将模型分成 4 部分，因而 `module_list` 中有 4 个子模型；将输入分割为 3 部分，则每次调用 `forward(x, split_sizes)`，函数内部的计算过程如下：

```
step=0     x0, x1, x2  |m0|    |m1|    |m2|    |m3|

step=1     x0, x1      |m0| x2 |m1|    |m2|    |m3|

step=2     x0          |m0| x1 |m1| x2 |m2|    |m3|

step=3                 |m0| x0 |m1| x1 |m2| x2 |m3|

step=4                 |m0|    |m1| x0 |m2| x1 |m3| x2

step=5                 |m0|    |m1|    |m2| x0 |m3| x1, x2

step=6                 |m0|    |m1|    |m2|    |m3| x0, x1, x2
```

不使用流水线，则任何时刻只有一个 GPU 在运行，而其他 GPU 则在等待这个 GPU 的数据；而使用流水线，例如上面计算过程中的 `step=3` 到 `step=4`，尽管在代码的写法为顺序执行：

```
x0 = m1(x0)
x1 = m2(x1)
x2 = m3(x2)
```

但由于 PyTorch 优秀的特性，上面的 3 行代码实际上是并行执行的，因为这 3 个在 CUDA 上的计算使用各自的数据，互不影响。

我们将之前的 CIFAR10 代码更改为多 GPU 流水线形式，修改后的代码位于 `SpikingFlow.softbp.examples.cifar10.py`。它的内容与 `SpikingFlow.softbp.examples.cifar10.py` 基本类似，我们只看主要的改动部分。

模型的定义，直接继承了 `ModelPipeline`。将模型拆成了 5 个部分，由于越靠前的层，输入的尺寸越大，越消耗显存，因此前面的少部分层会直接被单独分割出，而后面的很多层则放到了一起。需要注意的是，每次训练后仍然要重置 LIF 神经元的电压，因此要额外写一个重置函数 `reset_()`：

```
class Net(softbp.ModelPipeline):
    def __init__(self, gpu_list, tau=100.0, v_threshold=1.0, v_reset=0.0):
        super().__init__()
        # 网络结构，卷积-卷积-最大池化堆叠，最后接一个全连接层

        self.append(
            nn.Sequential(
                nn.Conv2d(3, 256, kernel_size=3, padding=1),
                nn.BatchNorm2d(256)
            ),
```

(下页继续)

(续上页)

```
            gpu_list[0]
        )

        self.append(
            nn.Sequential(
                neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset)
            ),
            gpu_list[1]
        )

        self.append(
            nn.Sequential(
                nn.Conv2d(256, 256, kernel_size=3, padding=1),
                nn.MaxPool2d(2, 2),
                nn.BatchNorm2d(256)
            ),
            gpu_list[2]
        )

        self.append(
            nn.Sequential(
                neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset)   #␣
↪16 * 16
            ),
            gpu_list[3]
        )

        self.append(
            nn.Sequential(
                nn.Conv2d(256, 256, kernel_size=3, padding=1),
                nn.BatchNorm2d(256),
                neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset),
                nn.Conv2d(256, 256, kernel_size=3, padding=1),
                nn.MaxPool2d(2, 2),
                nn.BatchNorm2d(256),
                neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset),   #␣
↪8 * 8
                nn.Conv2d(256, 256, kernel_size=3, padding=1),
                nn.BatchNorm2d(256),
                neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset),
                nn.Conv2d(256, 256, kernel_size=3, padding=1),
                nn.MaxPool2d(2, 2),
                nn.BatchNorm2d(256),
```

(下页继续)

(续上页)

```
                neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset),  #␣
↪4 * 4
                nn.Flatten(),
                nn.Linear(256 * 4 * 4, 10, bias=False),
                neuron.LIFNode(tau=tau, v_threshold=v_threshold, v_reset=v_reset)
            ),
            gpu_list[4]
        )

    def reset_(self):
        for item in self.modules():
            if hasattr(item, 'reset'):
                item.reset()
```

运行这份代码，由于分割的第 0 部分和第 3 部分占用的显存较小，因此将它们全部放在 0 号 GPU 上，而其他部分则各独占一个 GPU：

```
(pytorch-env) wfang@pami:~/SpikingFlow$ python ./SpikingFlow/softbp/examples/
↪cifar10mp.py
输入使用的 5 个 gpu，例如“0,1,2,0,3”  0,1,2,0,3
输入保存 CIFAR10 数据集的位置，例如“./”  ./tempdir
输入 batch_size，例如“64”  64
输入 split_sizes，例如“16”  4
输入学习率，例如“1e-3”  1e-3
输入仿真时长，例如“50”  50
输入 LIF 神经元的时间常数 tau，例如“100.0”  100.0
输入训练轮数，即遍历训练集的次数，例如“100”  100
输入保存 tensorboard 日志文件的位置，例如“./”  ./tempdir
```

稳定运行后，查看各个 GPU 显存的占用：

```
+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      4465      C   python                                     5950MiB |
|    1      4465      C   python                                     9849MiB |
|    2      4465      C   python                                     9138MiB |
|    3      4465      C   python                                     8936MiB |
+-----------------------------------------------------------------------------+
```

对于模型的不同分割方法会造成不同的显存占用情况。建议首先做一个简单的分割，然后用很小的 batch_size 和 split_sizes 去运行，再检查各个 GPU 显存的负载是否均衡，根据负载情况来重新调整分割。

分割后的模型，`batch_size=64, split_size=4`，根据 tensorboard 的记录显示，在 Tesla K80 上 30 分钟训练了 116 次；使用其他相同的参数，令 `batch_size=64, split_size=2`，30 分钟训练了 62 次；令 `batch_size=64, split_size=32`，30 分钟训练了 272 次；令 `batch_size=64, split_size=16`，30 分钟训练了 230 次；令 `batch_size=32, split_size=8`，30 分钟训练 335 次；令 `batch_size=32, split_size=16`，30 分钟训练 460 次；令 `batch_size=32, split_size=32`，30 分钟训练 466 次；不使用模型流水线、完全在同一个 GPU 上运行的 `SpikingFlow.softbp.examples.cifar10.py`，`batch_size=16`，30 分钟训练 759 次。对比如下表所示:

*.py	batch_size	split_size	images/minute
cifar10mp.py	64	32	580
	64	16	490
	64	4	247
	64	2	132
	32	8	357
	32	16	490
	32	32	497
cifar10.py	16	\	404

可以发现，参数的选择对于训练速度至关重要。合适的参数，例如 `batch_size=64, split_size=32`，训练速度已经明显超过单卡运行了。

3.1.2.6.7 持续恒定输入的更快方法

上文的代码中，我们使用泊松编码器 `encoding.PoissonEncoder()`，它以概率来生成脉冲，因此在不同时刻，这一编码器的输出是不同的。如果我们使用恒定输入，则每次前向传播时，不需要再重新启动一次流水线，而是可以启动一次流水线并一直运行，达到更快的速度、更小的显存占用。对于持续 `T` 的恒定输入 `x`，可以直接调用 `ModelPipeline.constant_forward(self, x, T, reduce)` 进行计算。

我们将之前的代码进行更改，去掉编码器部分，将图像数据直接送入 SNN。在这种情况下，我们可以认为 SNN 的第一个卷积层起到了“编码器”的作用：它接收图像作为输入，并输出脉冲。这种能够参与训练的编码器，通常能够起到比泊松编码器更好的编码效果，最终网络的分类性能也会有一定的提升。更改后的代码位于 `SpikingFlow.softbp.examples.cifar10cmp.py`，代码的更改非常简单，主要体现在:

```
class Net(softbp.ModelPipeline):
    ...
    # 使用父类的 constant_forward 来覆盖父类的 forward
    def forward(self, x, T):
        return self.constant_forward(x, T, True)
    ...

def main():
```

(下页继续)

(续上页)

```
    ...
    # 直接将图像送入网络，不需要编码器
    out_spikes_counter_frequency = net(img, T) / T
    ...
```

设置 batch_size=32，模型在显卡上的分布与之前相同，30 分钟训练 715 次；去掉编码器但不使用 ModelPipeline.constant_forward(self, x, T, reduce)，batch_size=64, split_size=32，30 分钟训练 276 次。可以发现，去掉编码器后网络的训练速度会变慢；使用这一方法能够起到一倍以上的加速。

3.1.2.7 事件驱动 SpikingFlow.event_driven

本教程作者：fangwei123456

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

需要注意的是，SpikingFlow.event_driven 与 SpikingFlow.softbp 类似，也是一个相对独立的包，与其他的 SpikingFlow.* 中的神经元、突触等组件不能混用。

3.1.2.7.1 事件驱动的 SNN 仿真

SpikingFlow.softbp 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())
    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。

3.1.2.7.2 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。

3.1.2.7.3 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。

3.1.2.7.4 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：

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

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

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

3.1.2.7.5 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 `SpikingFlow/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。

3.1.2.7.6 识别 MNIST

我们使用 Tempotron 搭建一个简单的 SNN 网络，识别 MNIST 数据集。首先我们需要考虑如何将 MNIST 数据集转化为脉冲输入。在“SpikingFlow.softbp“中我们习惯于使用泊松编码器，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。

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

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

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

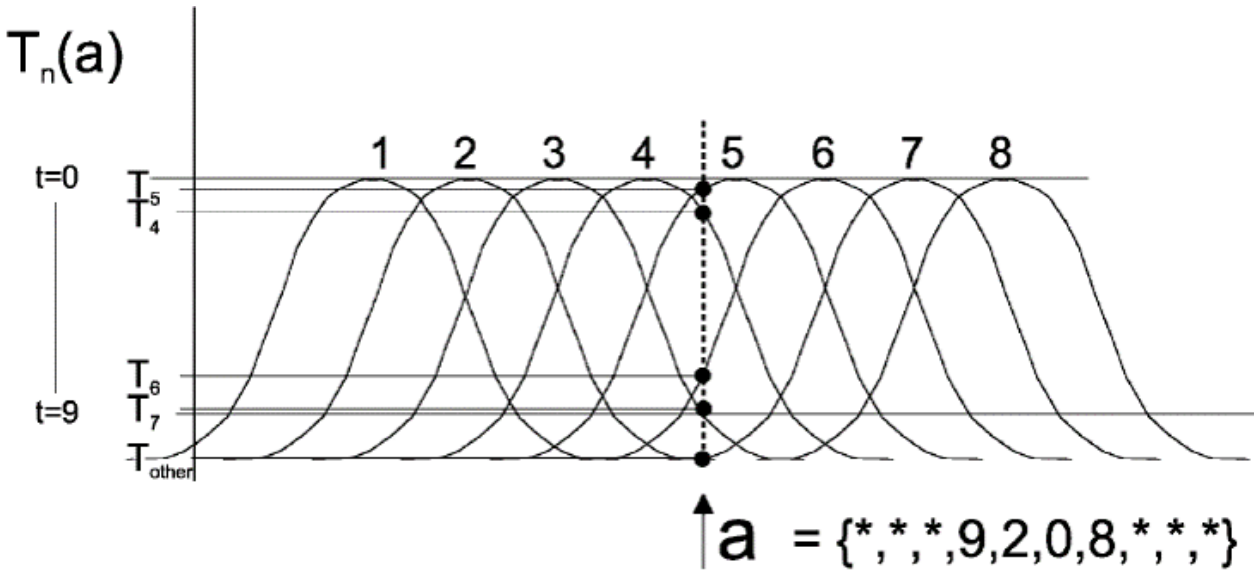
对于要编码的数据 $x \in X^i$，首先计算出 x 对应的高斯函数值 $g^i_j(x)$，这些函数值全部介于 $[0, 1]$ 之间。接下来，将函数值线性地转换到 $[0, T]$ 之间的脉冲发放时刻，其中 T 是编码周期，或者说是仿真时长：

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

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

形象化的示例如下图[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。

[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.



定义网络、损失函数、分类结果

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

```
class Net(nn.Module):
    def __init__(self, m, T):
        # m 是高斯调谐曲线编码器编码一个像素点所使用的神经元数量
        super().__init__()
        self.tempotron = neuron.Tempotron(784*m, 10, T)
    def forward(self, x: torch.Tensor):
        # 返回的是输出层 10 个 Tempotron 在仿真时长内的电压峰值
        return self.tempotron(x, 'v_max')
```

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

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

我们使用的损失函数与[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 `SpikingFlow.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\
```

(下页继续)

(续上页)

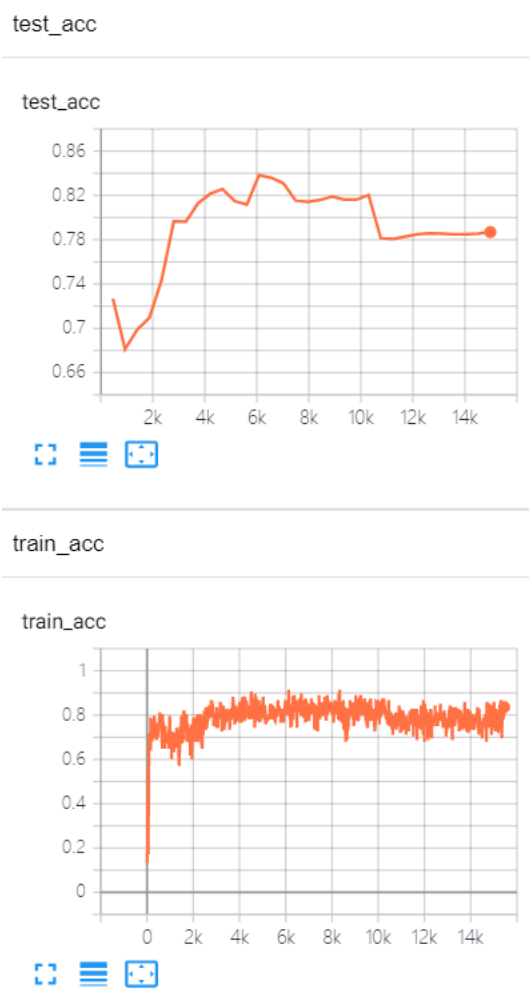
```
        同。shape=[batch_size, out_features] 的 tensor
        :param v_threshold: Tempotron 的阈值电压，float 或 shape=[batch_size, out_
↪features] 的 tensor
        :param label: 样本的真实标签，shape=[batch_size] 的 tensor
        :param num_classes: 样本的类别总数，int
        :return: 分类错误的神经元的电压，与阈值电压之差的均方误差
        '''
        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 SpikingFlow/event_driven/examples/tempotron-mnist.py：

```
(pytorch-env) wfang@e3:~/SpikingFlow$ python ./SpikingFlow/event_driven/examples/
↪tempotron-mnist.py
输入运行的设备，例如“cpu”或“cuda:0”   cuda:15
输入保存 MNIST 数据集的位置，例如“./”   ./dataset
输入 batch_size，例如“64”   128
输入学习率，例如“1e-3”   1e-3
输入仿真时长，例如“100”   100
输入训练轮数，即遍历训练集的次数，例如“100”   100
输入使用高斯调谐曲线编码每个像素点使用的神经元数量，例如“16”   16
输入保存 tensorboard 日志文件的位置，例如“./”   ./logs/tempotron-mnist
```

查看训练结果

Tensorboard 记录的训练结果如下：



测试集的正确率在 80% 左右，可以看出 Tempotron 确实实现了类似 ANN 中感知器的功能，具有一定的分类能力。

CHAPTER 4

文档索引

- genindex
- modindex
- search

CHAPTER 5

项目信息

SpikingFlow 由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 开发。

SpikingFlow 目前还在起步阶段，因此许多功能还不够完善。

Python 模块索引

S

索引

S

T

U

W

◆

模块