**Networks**

network继承的是pytorch里面的module

network下的方法：

**class** **Network**(torch**.**nn**.**Module):

**def** \_\_init\_\_(self, dt**=**1.0, batch\_size**=**1, learning**=**True, reward\_fn **=** None):

*#这learning表示的是是否允许connection update*

*# reward\_fn设计到强化学习里的reward函数*

**...**

**def** **add\_layer**(self, layer:str):

self**.**layers[name] **=** layer

self**.**add\_module(name, layer)

layer**.**train(self**.**learning)

layer**.**compute\_decays(self**.**dt)

layer**.**set\_batch\_size(self**.**batch\_size)

**def** **add\_connection**(self, str,str):

self**.**connections **...**

connection**.**train(self**.**learning)

**def** **add\_monitor**(self,str):

**...**

**def** **save**(self, str):

torch**.**save(self, open(file\_name, "wb"))

**def** **clone**(self,):

**pass**

搭建好网络就可以run网络了：

**def** **run**(self, inputs, time, one\_step**=**False):

timesteps **=** int(time **/** self**.**dt)*#总模拟步长*

*#这个one step指的是是否一个time step跑完所有的层，这些层是按照添加的顺序跑*

*#一般是默认不是one step*

**for** t **in** range(timesteps):

current\_inputs **=** {}

**if** **not** one\_step:*#如果不是一次性跑完*

*#就分别得到每一层的输入是什么*

*# {'X': tensor(), 'Ae': tensor(), 'Ai': tensor()}*

current\_inputs**.**update(self**.**\_get\_inputs())

*#得到了每一层的输入*

*#其实因为网络，初始化的时候 每一层的spike初始化为0*

*#按照此时的计算的方式，此时得到的输入依旧为0*

*#这个是网络没有外部输入的时候，整个网络在内部驱动下跑*

**for** l **in** self**.**layers: *#l是layer的名字 X Ae Ai之类*

**if** l **in** inputs: *#如果是输入里的比如X*

**if** l **in** current\_inputs:*#已经在current input里 就加上去*

current\_inputs[l] **+=** inputs[l][t]

**else**

current\_inputs[l] **=** inputs[l][t]

*#其实没有上述的那么复杂，就是，对于输入层*

*#current\_inputs [l] += inputs[l][t]*

self**.**layers[l]**.**forward(x**=**current\_inputs[l])*#得到该层的输入并得到输出*

*#利用STDP更新权重*

**for** c **in** self**.**connections:

self**.**connesctions[c]**.**update(leanring **=** self**.**learning)

*#再次计算得到输入，个人感觉此时这边有点重复了，没看出来*

*#这边再次计算得到输入有什么作用*

current\_inputs**.**update(self**.**\_get\_inputs())

*#record monitor*

**for** m **in** self**.**monitors:

self**.**monitors[m]**.**record()

*#所有的时间跑完再normalize一下*

**for** c **in** self**.**connections:

self**.**connections[c]**.**normalize()

总的来看，这个函数就是依赖connection连接的source的脉冲，计算此时target layer的输入是什么，这样就得到了每一层的输入，也就与上面的对应上了。

# Nodes

Node大概就是定义网络一个layer时，用哪一类的神经元来定义这一层，比如说有

* Input 对应的是输入层
* McCullochPitts 对应的一般的ANN里的神经元
* IF Nodes
* LIF Nodes
* CurrentLIFNodes
* AdaptiveLIFNodes
* DiehlAndCookNodes
* Izhikevich Nodes
* SRM0Nodes

所有Nodes的基类：

对于一个结点非常重要的三个值：

* self.x 脉冲 trace
* self.s 脉冲
* self.v 电压

**class** **Nodes**(torch**.**nn**.**Module)：

**def** \_\_init\_\_(self, n, shape, traces**=**False, traces\_additive**=**False, tc\_trace**=**False, trace\_scale, sum\_input, learning**=**True):

*#这个n是这一层的神经元的个数*

*#shape是这一层的维度*

*#traces是是否记录spike traces*

*#traces\_additive是否额外加上去spike trace*

*#tc\_trace 时间常数 trace\_scale是spike trace的缩放系数*

**pass**

*#看下node的前馈过程*

**def** **forward**(self, x):

*#这边用到trace*

**if** self**.**traces:

*#首先把trace decay一下 这个tracedecay就是一个指数下降*

self**.**x **\*=** self**.**trace\_decay *#这边的x就是node的trace*

**if** self**.**traces\_additive: *#如果在之间的trace上继续用就+1*

self**.**x **+=** self**.**trace\_scale **\*** self**.**s**.**float()

**else**:*#如果不在上面加 就是spike的地方trace为1*

self**.**x**.**masked\_fill\_(self**.**s **!=** 0, 1)

**def** **reset\_state\_variables**(self):

self**.**s**.**zero\_() *#reset spike*

self**.**x**.**zero\_() *#reset trace*

self**.**summed**.**zero\_() *#reset sum input 一般用不到*

**def** **compute\_decays**(self, dt):

self**.**trace\_decay **=** torch**.**exp(**-**self**.**dt **/** self**.**tc\_trace)

**def** **set\_batch\_size**(self, batch\_size):

*#感觉初始化了脉冲， trace 以及summedinput*

self**.**s **=** torch**.**zeros()

self**.**x **=** torch**.**zeros()

self**.**summed **=** torch**.**zeros()

**def** **train**(self, mode:True):

self**.**training **=** mode

**return** super()**.**train(mode)

主要是在操作这个trace。这个spike trace说的是，当一个神经元发生脉冲的时候，在之后的时间内，并不是一下子变为0，而是一个指数缩减。

**InputNode：**

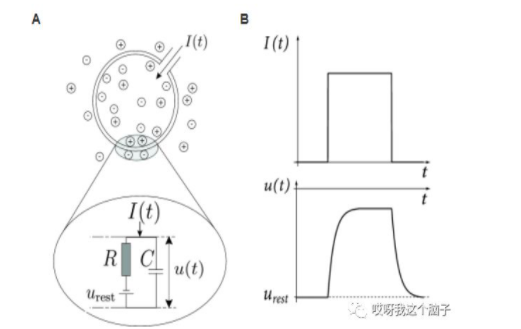
对于input节点，这边假设的是，输入就是输出的脉冲，主要是对于forward函数修改，然后剩下和基类里面一样

**def** **forward**(self, x):

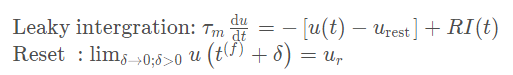
self**.**s **=** x

super()**.**forward(x)

对于LIF节点，可以看到，对于电压的计算，对应到lif公式

LIF节点。

<https://blog.csdn.net/ly18846826264/article/details/103402621>



Leaky指泄露，表示如果神经元输入只有一个时，不足以让膜电势超过阈值，由于细胞膜不断进行膜内外离子交换，膜电势会自动发生泄露逐渐回落到静息状态；

Integrate指积分，表示神经元会接收所有与该神经元相连的轴突末端（上一个神经元）到来的脉冲；

Fire指激发，表示当膜电势超过阈值时，神经元会发送脉冲。神经元发送脉冲后会进入超极化状态，然后是不应期（Refractory Period），在不应期内即使给予刺激也不会反应，即神经元不再接收刺激，保持静息电位。

**def** **forward**(self, x):

self**.**v **=** self**.**decay **\*** (self**.**v **-** self**.**reset) **+** self**.**reset

self**.**v **+=** (self**.**refrac\_count **==** 0)**.**float() **\*** x)*#不在不应期的那部分增加*

*#然后更新不应期*

self**.**refrac\_count **=** (self**.**refrac\_count **>** 0)**.**float() **\*** (self**.**refrac\_count **-** self**.**dt)

*#超过阈值的spike*

self**.**s **=** self**.**v **>=** self**.**thresh

*#发放脉冲的进入不应期，且电压reset*

self**.**refrac\_count**.**masked\_fill\_(self**.**s, self**.**refrac)

self**.**v**.**masked\_fill\_(self**.**s, self**.**reset)

*#clip到lower bound*

super()**.**forward(x)

**def** **set\_batch\_size**(self, batch\_size):

super()**.**set\_batch\_size(batch\_size**=**batch\_size)

self**.**v **=** self**.**reset **\*** torch**.**ones(**...**)

self**.**refrac\_count **=** torch**.**zeros\_like(self**.**v, device**..**)

关于Diehl老哥的节点，就是比一般的LIF多了个adaptive threshold，这个**自适应阈值**说的什么呢，就是，当一个地方老是发spike的时候，就给他**阈值加点**，这样就很难发脉冲了，然后**不发spike了，就把这个阈值指数衰减一点**，剩下的和lif就一样了

# Connection

Node形成的layer和layer之间该怎么连接，比如有FC层，conv层，pooling层之类。

connection基类：

**class** **AbstractConnection**(ABC, Module):

**def** \_\_init\_\_(self, source, target, nu, reduction, weight\_decay):

'''

source 和 target就是字面意思 起始连接 目标连接

nu是学习率

reduction是你缩减的方式是用mean还是max

weight\_decay是你对于weight是否衰减 和 普通的ANN没什么区别

'''

**if** self**.**update\_rule **is** None:

self**.**update\_rule **=** NoOp

self**.**update\_rule **=** self**.**update\_rule(connection**=**self, nu**=**nu, reduction**=**reduction, weight\_decay**=**weight\_decay)

这边就只看一下全连接，主要看一下这边的compute是咋搞的，其实就是传统的wx+b

**def** **compute**(self, s):

post **=** s**.**float()**.**view(s**.**size(0), **-**1) @ self**.**w **+** self**.**b

**return** post**.**view(s**.**size(0), **\***self**.**target**.**shape)

这边的compute主要是计算，上一层的脉冲，经过加权和之后传递到下一层

这里的update主要是更新connection里的weight，可以看到主要是依靠update\_rule来更新。

# Learning

Weight的更新方式：

传统的人工神经网络是依靠反向传播算法，而SNN由于不可微分的特性，所以更多的利用STDP之类的突触可塑性法则。

基类里面对于update时，主要是对于weight进行decay，以及把weight clamp到一个界里面

**def** **update**(self):

**if** self**.**weight\_decay:

self**.**connection**.**w **-=** self**.**weight\_decay **\*** self**.**connection**.**w

**if** **not** isinstance(self, NoOp):

self**.**connection**.**w**.**clamp\_(self**.**connection**.**wmin, self**.**connection**.**wmax)

STDP：



**def** **\_connection\_update**(self):

batch\_size **=** **...**

source\_s **=** **...**

source\_x **=** **...**

target\_s **=** **...**

target\_x **=** **..**

update1 **=** torch**.**mean(torch**.**bmm(source\_s, target\_x), dim**=**0)

update2 **=** torch**.**mean(torch**.**bmm(source\_x, target\_s), dim**=**0)

self**.**connection**.**w **=** self**.**connection**.**w **-** self**.**nu[0]**\***update1 **+** self**.**nu[1]**\***update2

WeightDependentPostPre指的update是依赖于weight的幅度

# Encoding

这个主要是对于SNN的输入的编码的方式，一般ANN的输入的时候，在pytorch里面

train\_dset **=** mnist**.**MNIST(root**=**'./mnist\_data', train**=**True,

download**=**True,

transform**=**img\_transform,

target\_transform**=**OneHot(10),

)

val\_dset **=** mnist**.**MNIST(root**=**'./mnist\_data', train**=**False,

download**=**True,

transform**=**img\_transform,

target\_transform**=**OneHot(10),

)

train\_loader **=** DataLoader(train\_dset, batch\_size**=**BATCH\_SIZE,

shuffle**=**True)

val\_loader **=** DataLoader(val\_dset, batch\_size**=**BATCH\_SIZE,

shuffle**=**False)

**for** imgs, labels **in** train\_loader:

**....**

这边对于输入的dataloadr先编码了一下，主要写在datasets里的torchvision\_wrapper里面，继承的是torchvision里的datasets，主要看一下get\_items部分

**def** \_\_getitem\_\_(self, ind):

image, label **=** super()**.**\_\_getitem\_\_(ind)

output **=** {

"image": image,

"label": label,

"encoded\_image": self**.**image\_encoder(image)

"encoded\_label": self**.**label\_encoder(label)

}

**return** output

传回的数据是对于原来的image和label encode之后的，这边的label\_encoder是NoOp就是和原来一样。然后encode的方式实际上就是将pixel的大小，变成了发放概率。（输入值变成频率）

然后拿possion分布（编码方式）举例。

**def** **poisson**(datum, time, dt):

shape, size **=** datum**.**shape, datum**.**numel()

datum **=** datum**.**flatten() #按照行降维

time **=** int(time**/**dt)

rate **=** torch**.**zeros(size)

rate[datum **!=** 0] **=** 1 **/** datum[datum **!=** 0] **\*** (1000 **/** dt)*#firing rates*

dist **=** torch**.**distributions**.**Poisson(rate**=**rate)

intervals **=** dist**.**sample(sample\_shape**=**torch**.**Size([time **+** 1]))

intervals[:, datum **!=** 0] **+=** (intervals[:, datum **!=** 0] **==** 0)**.**float()

times **=** torch**.**cumsum(intervals, dim**=**0)**.**long()

times[times **>=** time **+** 1] **=** 0

spikes **=** torch**.**zeros(time **+** 1, size)**.**byte()

spikes[times, torch**.**arange(size)] **=** 1

spikes **=** spikes[1:]

**return** spikes**.**view(time, **\***shape)

# Evaluation

分类以及给每个神经元赋予label

首先看all\_activity，传入的参数是最后一层的输出spike，每个神经元的assignment。

**def** **all\_activity**():

*# spike的shape 为 n\_sample\*time\*n\_neurons*

n\_samples **=** spikes**.**size(0)

spikes **=** spikes**.**sum(1)*#沿着时间维度求和*

rates **=** torch**.**zeros(n\_samples, 10)

**for** i **in** range(10):

n\_assigns **=** torch**.**sum(assignments**=**i)**.**float()*#看一下为第i类的神经元数量*

**if** n\_assigns **>** 0:

indices **=** torch**.**nonzero(assignmnts**==**i)**.**view(**-**1)看一下哪些神经元的index为i

*#然后找到对应该类的所有的spikes求个平均*

就得到平均spikes

rates[:,i] **=** torch**.**sum(spikes[:, indices],1) **/** n\_assigns

*#这样就计算出了十类，每一类的平均spikes为多少*

*#然后按照降序排列*

*#第一的那个就是该样本的输出*

**return** torch**.**sort(rates, dim**=**1, descending**=**True)[1][:,0]

在测试的时候，主要是要把训练时的assignment保留下来，下面看一下怎么计算出这个assignment，也就是说，怎么知道每一个神经元到底对应着是哪一类。输入是最后一层的spikes，以及真实的label， assignments 是 一个n\_neuron维度的tensor，便是的是每个神经元被赋予的label。

**def** **assignmetns**():

n\_neurons **=** spikes**.**size(2)

**if** rates **is** None:

*#这个rates如果被传入的话*

*#就说明是从上一次assignment label里得到的接着传入*

rates **=** torch**.**zeros(n\_neurons, 10)

*#沿着时间维度求和 n\_sample \* time \* n\_neuron*

spikes **=** spikes**.**sum(1)

*# n\_sample \* n\_neuron*

*#torch.sum(spikes[indices],0) 计算得到的是对应着该label的每个神经元的spike的总数*

**for** i **in** range(10):

n\_labeled **=** torch**.**sum(labels**==**i)**.**float()

*#计算该label的数量*

**if** n\_labeled **>** 0:

*#看下哪些sample对应着该label*

indices **=** torch**.**nonzero(labels**==**i)**.**view(**-**1)

rates[:,i] **=** alpha **\*** rates[:,i] **+** (torch**.**sum(spikes[indices],0)**/**n\_labeled)*#计算平均rate*

*#这边相当于先归一化一下*

proportions **=** rates **/** rates**.**sum(1, keepdim**=**True)

proportions[proportions **!=** proportions] **=** 0 *# Set NaNs to 0*

*#这个proportions本来计算的是n\_neurons \* 10*

*#然后看了一下哪一个label对应的高 就是哪一类*

assignments **=** torch**.**max(proportions, 1)[1]

可以看到就是得到输出的spikes，在时间维度上求和得到



然后找到对应label是i的sample，比如说是第1 3 4 6 7个sample，然后，对这五个sample这N个neuron的spike求个平均，就得到label为i时，每个neuron对应的spike的数量，这些东西存到rates里，即rates为



就是说，每个neuron，都有着十类的spikes的输出，然后归一化一下，然后看一下这个Neuron对应的十类的平均spikes，哪一类的最多，该神经元就被assign成哪一类的label