
SLAYER PyTorch

Release 0.1

Sumit Bam Shrestha

Jun 26, 2019

CONTENTS:

1	SLAYER PyTorch main	1
2	SLAYER module	3
3	Spike Classifier	7
4	Spike Loss	9
5	Spike Input/Output	11
6	Learning statistics	17
7	Optimizer	21
8	Indices and tables	23
8.1	Usage:	23
8.2	Example:	23
	Python Module Index	25
	Index	27

SLAYER PYTORCH MAIN

This module bundles various SLAYER PyTorch modules as a single package. The complete module can be imported as

```
>>> import slayerSNN as snn
```

- The **spike-layer** module will be available as `snn.layer`.
- The **yaml-parameter** module will be available as `snn.params`.
- The **spike-loss** module will be available as `snn.loss`.
- The **spike-classifier** module will be available as `snn.predict`.
- The **spike-IO** module will be available as `snn.io`.

SLAYER MODULE

class `slayer.spikeLayer` (*neuronDesc*, *simulationDesc*, *fullRefKernel=False*)

This class defines the main engine of SLAYER. It provides necessary functions for describing a SNN layer. The input to output connection can be fully-connected, convolutional, or aggregation (pool). It also defines the psp operation and spiking mechanism of a spiking neuron in the layer.

Important: It assumes all the tensors that are being processed are 5 dimensional. (Batch, Channels, Height, Width, Time) or NCHWT format. The user must make sure that an input of correct dimension is supplied.

*If the layer does not have spatial dimension, the neurons can be distributed along either Channel, Height or Width dimension where Channel * Height * Width is equal to number of neurons. It is recommended (for speed reasons) to define the neurons in Channels dimension and make Height and Width dimension one.*

Arguments:

- **neuronDesc** (`slayer.yamlParams`): spiking neuron descriptor.

```
neuron:
  type:      SRMALPHA  # neuron type
  theta:     10         # neuron threshold
  tauSr:     10.0       # neuron time constant
  tauRef:    1.0        # neuron refractory time constant
  scaleRef:  2          # neuron refractory response scaling (relative
↪to theta)
  tauRho:    1          # spike function derivative time constant
↪(relative to theta)
  scaleRho:  1          # spike function derivative scale factor
```

- **simulationDesc** (`slayer.yamlParams`): simulation descriptor

```
simulation:
  Ts: 1.0
  tSample: 300
  nSample: 12
```

- **fullRefKernel** (bool, optional): high resolution refractory kernel (the user shall not use it in practice)

Usage:

```
>>> snnLayer = slayer.spikeLayer(neuronDesc, simulationDesc)
```

conv (*inChannels*, *outChannels*, *kernelSize*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *weightScale=100*)

Returns a function that can be called to apply conv layer mapping to input tensor per time instance. It behaves same as `torch.nn.conv2d` applied for each time instance.

Arguments:

- `inChannels (int)`: number of channels in input
- `outChannels (int)`: number of channels produced by convolution
- `kernelSize (int or tuple of two ints)`: size of the convolving kernel
- `stride (int or tuple of two ints)`: stride of the convolution. Default: 1
- `padding (int or tuple of two ints)`: zero-padding added to both sides of the input. Default: 0
- `dilation (int or tuple of two ints)`: spacing between kernel elements. Default: 1
- `groups (int or tuple of two ints)`: number of blocked connections from input channels to output channels. Default: 1

The parameters `kernelSize`, `stride`, `padding`, `dilation` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of two `ints` – in which case, the first `int` is used for the height dimension, and the second `int` for the width dimension

Usage:

```
>>> conv = snnLayer.conv(2, 32, 5) # 32C5 filter
>>> output = conv(input)           # must have 2 channels
```

delay (inputSize)

Returns a function that can be called to apply delay operation in time dimension of the input tensor. The `delay` parameter is available as `delay.delay` and is initialized uniformly between 0ms and 1ms. The `delay` parameter is stored as float values, however, it is floored during actual delay application internally. The delay values are not clamped to zero. To maintain the causality of the network, one should clamp the delay values explicitly to ensure positive delays.

Arguments:

- `inputSize (int or tuple of three ints)`: spatial shape of the input signal in CHW format (Channel, Height, Width). If integer value is supplied, it refers to the number of neurons in channel dimension. Height and Width are assumed to be 1.

Usage:

```
>>> delay = snnLayer.delay((C, H, W))
>>> delayedSignal = delay(input)
```

Always clamp the delay after `optimizer.step()`.

```
>>> optimizer.step()
>>> delay.delay.data.clamp_(0)
```

delayShift (input, delay, Ts=1)

Applies delay in time dimension (assumed to be the last dimension of the tensor) of the input tensor. The autograd backward link is established as well.

Arguments:

- `input`: input Torch tensor.
- `delay (float or Torch tensor)`: amount of delay to apply. Same delay is applied to all the inputs if `delay` is `float` or Torch tensor of size 1. If the Torch tensor has size more than 1, its dimension must match the dimension of input tensor except the last dimension.

- Ts: sampling time of the delay. Default is 1.

Usage:

```
>>> delayedInput = slayer.delayShift(input, 5)
```

dense (*inFeatures, outFeatures, weightScale=10*)

Returns a function that can be called to apply dense layer mapping to input tensor per time instance. It behaves similar to `torch.nn.Linear` applied for each time instance.

Arguments:

- *inFeatures* (*int*, tuple of two ints, tuple of three ints): dimension of input features (Width, Height, Channel) that represents the number of input neurons.
- *outFeatures* (*int*): number of output neurons.

Usage:

```
>>> fcl = snnLayer.dense(2048, 512)           # takes (N, 2048, 1, 1, T) tensor
>>> fcl = snnLayer.dense((128, 128, 2), 512) # takes (N, 2, 128, 128, T)
↳ tensor
>>> output = fcl(input)                       # output will be (N, 512, 1, 1,
↳ T) tensor
```

dropout (*p=0.5, inplace=False*)

Returns a function that can be called to apply dropout layer to the input tensor. It behaves similar to `torch.nn.Dropout`. However, dropout over time dimension is preserved, i.e. if a neuron is dropped, it remains dropped for entire time duration.

Arguments:

- *p*: dropout probability.
- *inplace* (*bool*): inplace operation flag.

Usage:

```
>>> drop = snnLayer.dropout(0.2)
>>> output = drop(input)
```

pool (*kernelSize, stride=None, padding=0, dilation=1*)

Returns a function that can be called to apply pool layer mapping to input tensor per time instance. It behaves same as `torch.nn.sum` pooling applied for each time instance.

Arguments:

- *kernelSize* (*int* or tuple of two ints): the size of the window to pool over
- *stride* (*int* or tuple of two ints): stride of the window. Default: *kernelSize*
- *padding* (*int* or tuple of two ints): implicit zero padding to be added on both sides. Default: 0
- *dilation* (*int* or tuple of two ints): a parameter that controls the stride of elements in the window. Default: 1

The parameters *kernelSize*, *stride*, *padding*, *dilation* can either be:

- a single *int* – in which case the same value is used for the height and width dimension
- a tuple of two ints – in which case, the first *int* is used for the height dimension, and the second *int* for the width dimension

Usage:

```
>>> pool = snnLayer.pool(4) # 4x4 pooling
>>> output = pool(input)
```

psp (*spike*)

Applies psp filtering to spikes. The output tensor dimension is same as input.

Arguments:

- spike: input spike tensor.

Usage:

```
>>> filteredSpike = snnLayer.psp(spike)
```

pspLayer ()

Returns a function that can be called to apply psp filtering to spikes. The output tensor dimension is same as input. The initial psp filter corresponds to the neuron psp filter. The psp filter is learnable. NOTE: the learned psp filter must be reversed because PyTorch performs correlation operation.

Usage:

```
>>> pspLayer = snnLayer.pspLayer()
>>> filteredSpike = pspLayer(spike)
```

spike (*membranePotential*)

Applies spike function and refractory response. The output tensor dimension is same as input. *membranePotential* will reflect spike and refractory behaviour as well.

Arguments:

- membranePotential: subthreshold membrane potential.

Usage:

```
>>> outSpike = snnLayer.spike(membranePotential)
```

class `slayer.yamlParams` (*parameter_file_path*)

This class reads yaml parameter file and allows dictionary like access to the members.

Usage:

```
import slayerSNN as snn
netParams = snn.params('path_to_yaml_file') # OR
netParams = slayer.yamlParams('path_to_yaml_file')

netParams['training']['learning']['etaW'] = 0.01
print('Simulation step size', netParams['simulation']['Ts'])
print('Spiking neuron time constant', netParams['neuron']['tauSr'])
print('Spiking neuron threshold', netParams['neuron']['theta'])

netParams.save('filename.yaml')
```

SPIKE CLASSIFIER

class spikeClassifier.**spikeClassifier**

It provides classification modules for SNNs. All the functions it supplies are static and can be called without making an instance of the class.

static getClass (*spike*)

Returns the predicted class label. It assigns single class for the SNN output for the whole simulation runtime.

Usage:

```
>>> predictedClass = spikeClassifier.getClass(spikeOut)
```


SPIKE LOSS

class spikeLoss.**spikeLoss** (*networkDescriptor*, *slayerClass*=<class 'slayer.spikeLayer'>)

This class defines different spike based loss modules that can be used to optimize the SNN.

Usage:

```
>>> error = spikeLoss.spikeLoss(networkDescriptor)
>>> error = spikeLoss.spikeLoss(errorDescriptor, neuronDesc, simulationDesc)
```

numSpikes (*spikeOut*, *desiredClass*, *numSpikesScale*=1)

Calculates spike loss based on number of spikes within a *target region*. The *target region* and *desired spike count* is specified in `error.errorDescriptor['tgtSpikeRegion']` Any spikes outside the target region are penalized with `error.spikeTime` loss..

$$e(t) = \begin{cases} \frac{actualSpikeCount - desiredSpikeCount}{targetRegionLength} & \text{for } t \in targetRegion \\ (\varepsilon * (output - desired))(t) & \text{otherwise} \end{cases}$$
$$E = \int_0^T e(t)^2 dt$$

Arguments:

- `spikeOut` (`torch.tensor`): spike tensor
- `desiredClass` (`torch.tensor`): one-hot encoded desired class tensor. Time dimension should be 1 and rest of the tensor dimensions should be same as `spikeOut`.

Usage:

```
>>> loss = error.numSpikes(spikeOut, target)
```

spikeTime (*spikeOut*, *spikeDesired*)

Calculates spike loss based on spike time. The loss is similar to van Rossum distance between output and desired spike train.

$$E = \int_0^T (\varepsilon * (output - desired))(t)^2 dt$$

Arguments:

- `spikeOut` (`torch.tensor`): spike tensor
- `spikeDesired` (`torch.tensor`): desired spike tensor

Usage:

```
>>> loss = error.spikeTime(spikeOut, spikeDes)
```

SPIKE INPUT/OUTPUT

`spikeFileIO.encode1DnumSpikes` (*filename, nID, tSt, tEn, nSp*)

Writes binary spike file given a tuple specifying neuron, start of spike region, end of spike region and number of spikes.

The binary file is encoded as follows:

- Number of spikes data is represented by an 80 bit number
- First 16 bits (bits 79-64) represent the neuronID
- Next 24 bits (bits 63-40) represents the start time in microseconds
- Next 24 bits (bits 39-16) represents the end time in microseconds
- Last 16 bits (bits 15-0) represents the number of spikes

Arguments:

- `filename` (string): path to the binary file
- `nID` (numpy array): neuron ID
- `tSt` (numpy array): region start time (in milliseconds)
- `tEn` (numpy array): region end time (in milliseconds)
- `nSp` (numpy array): number of spikes in the region

Usage:

```
>>> spikeFileIO.encode1DnumSpikes(file_path, nID, tSt, tEn, nSp)
```

`spikeFileIO.encode1Dspikes` (*filename, TD*)

Writes one dimensional binary spike file from a TD event.

The binary file is encoded as follows:

- Each spike event is represented by a 40 bit number.
- First 16 bits (bits 39-24) represent the neuronID.
- Bit 23 represents the sign of spike event: 0=>OFF event, 1=>ON event.
- the last 23 bits (bits 22-0) represent the spike event timestamp in microseconds.

Arguments:

- `filename` (string): path to the binary file.
- `TD` (an `spikeFileIO.event`): TD event.

Usage:

```
>>> spikeFileIO.write1Dspikes(file_path, TD)
```

`spikeFileIO.encode2Dspikes(filename, TD)`

Writes two dimensional binary spike file from a TD event. It is the same format used in neuromorphic datasets NMNIST & NCALTECH101.

The binary file is encoded as follows:

- Each spike event is represented by a 40 bit number.
- First 8 bits (bits 39-32) represent the xID of the neuron.
- Next 8 bits (bits 31-24) represent the yID of the neuron.
- Bit 23 represents the sign of spike event: 0=>OFF event, 1=>ON event.
- The last 23 bits (bits 22-0) represent the spike event timestamp in microseconds.

Arguments:

- `filename (string)`: path to the binary file.
- `TD (an spikeFileIO.event)`: TD event.

Usage:

```
>>> spikeFileIO.write2Dspikes(file_path, TD)
```

`spikeFileIO.encode3Dspikes(filename, TD)`

Writes binary spike file for TD event in height, width and channel dimension.

The binary file is encoded as follows:

- Each spike event is represented by a 56 bit number.
- First 12 bits (bits 56-44) represent the xID of the neuron.
- Next 12 bits (bits 43-32) represent the yID of the neuron.
- Next 8 bits (bits 31-24) represents the channel ID of the neuron.
- The last 24 bits (bits 23-0) represent the spike event timestamp in microseconds.

Arguments:

- `filename (string)`: path to the binary file.
- `TD (an spikeFileIO.event)`: TD event.

Usage:

```
>>> spikeFileIO.write3Dspikes(file_path, TD)
```

class `spikeFileIO.event(xEvent, yEvent, pEvent, tEvent)`

This class provides a way to store, read, write and visualize spike event.

Members:

- `x (numpy int array)`: *x* index of spike event.
- `y (numpy int array)`: *y* index of spike event (not used if the spatial dimension is 1).
- `p (numpy int array)`: *polarity* or *channel* index of spike event.
- `t (numpy double array)`: *timestamp* of spike event. Time is assumend to be in ms.

Usage:


```
>>> TD = spikeFileIO.event(xEvent, yEvent, pEvent, tEvent)
```

toSpikeArray (*samplingTime=1, dim=None*)

Returns a numpy tensor that contains the spike events sampled in bins of *samplingTime*. The array is of dimension (channels, height, time) or “CHT” for 1D data. The array is of dimension (channels, height, width, time) or “CHWT” for 2D data.

Arguments:

- *samplingTime*: the width of time bin to use.
- *dim*: the dimension of the desired tensor. Assigns dimension itself if not provided.

Usage:

```
>>> spike = TD.toSpikeArray()
```

toSpikeTensor (*emptyTensor, samplingTime=1*)

Returns a numpy tensor that contains the spike events sampled in bins of *samplingTime*. The tensor is of dimension (channels, height, width, time) or “CHWT”.

Arguments:

- *emptyTensor* (numpy or torch tensor): an empty tensor to hold spike data
- *samplingTime*: the width of time bin to use.

Usage:

```
>>> spike = TD.toSpikeTensor( torch.zeros((2, 240, 180, 5000)) )
```

spikeFileIO.read1DnumSpikes (*filename*)

Reads a tuple specifying neuron, start of spike region, end of spike region and number of spikes from binary spike file.

The binary file is encoded as follows:

- Number of spikes data is represented by an 80 bit number.
- First 16 bits (bits 79-64) represent the neuronID.
- Next 24 bits (bits 63-40) represents the start time in microseconds.
- Next 24 bits (bits 39-16) represents the end time in microseconds.
- Last 16 bits (bits 15-0) represents the number of spikes.

Arguments:

- *filename* (string): path to the binary file

Usage:

```
>>> nID, tSt, tEn, nSp = spikeFileIO.read1DnumSpikes(file_path)
``tSt`` and ``tEn`` are returned in milliseconds
```

spikeFileIO.read1Dspikes (*filename*)

Reads one dimensional binary spike file and returns a TD event.

The binary file is encoded as follows:

- Each spike event is represented by a 40 bit number.
- First 16 bits (bits 39-24) represent the neuronID.

- Bit 23 represents the sign of spike event: 0=>OFF event, 1=>ON event.
- the last 23 bits (bits 22-0) represent the spike event timestamp in microseconds.

Arguments:

- `filename (string)`: path to the binary file.

Usage:

```
>>> TD = spikeFileIO.read1Dspikes(file_path)
```

`spikeFileIO.read2Dspikes (filename)`

Reads two dimensional binary spike file and returns a TD event. It is the same format used in neuromorphic datasets MNIST & NCALTECH101.

The binary file is encoded as follows:

- Each spike event is represented by a 40 bit number.
- First 8 bits (bits 39-32) represent the xID of the neuron.
- Next 8 bits (bits 31-24) represent the yID of the neuron.
- Bit 23 represents the sign of spike event: 0=>OFF event, 1=>ON event.
- The last 23 bits (bits 22-0) represent the spike event timestamp in microseconds.

Arguments:

- `filename (string)`: path to the binary file.

Usage:

```
>>> TD = spikeFileIO.read2Dspikes(file_path)
```

`spikeFileIO.read3Dspikes (filename)`

Reads binary spike file for spike event in height, width and channel dimension and returns a TD event.

The binary file is encoded as follows:

- Each spike event is represented by a 56 bit number.
- First 12 bits (bits 56-44) represent the xID of the neuron.
- Next 12 bits (bits 43-32) represent the yID of the neuron.
- Next 8 bits (bits 31-24) represents the channel ID of the neuron.
- The last 24 bits (bits 23-0) represent the spike event timestamp in microseconds.

Arguments:

- `filename (string)`: path to the binary file.

Usage:

```
>>> TD = spikeFileIO.read3Dspikes(file_path)
```

`spikeFileIO.showTD (TD, frameRate=24, preComputeFrames=True, repeat=False)`

Visualizes TD event.

Arguments:

- `TD`: spike event to visualize.
- `frameRate`: framerate of visualization.

- `preComputeFrames`: flag to enable precomputation of frames for faster visualization. Default is `True`.
- `repeat`: flag to enable repeat of animation. Default is `False`.

Usage:

```
>>> showTD(TD)
```

`spikeFileIO.spikeArrayToEvent` (*spikeMat*, *samplingTime=1*)

Returns TD event from a numpy array (of dimension 3 or 4). The numpy array must be of dimension (channels, height, time) or “CHT” for 1D data. The numpy array must be of dimension (channels, height, width, time) or “CHWT” for 2D data.

Arguments:

- `spikeMat`: numpy array with spike information.
- `samplingTime`: time width of each time bin.

Usage:

```
>>> TD = spikeFileIO.spikeArrayToEvent(spike)
```


LEARNING STATISTICS

class learningStats.learningStat

This class collect the learning statistics over the epoch.

Usage:

This class is designed to be used with learningStats instance although it can be used separately.

```
>>> trainingStat = learningStat()
```

accuracy()

Returns the average accuracy calculated from the point the stats was reset.

Usage:

```
>>> loss = trainingStat.accuracy()
```

loss()

Returns the average loss calculated from the point the stats was reset.

Usage:

```
>>> loss = trainingStat.loss()
```

reset()

Reset the learning staistics. This should usually be done before the start of an epoch so that new statistics counts can be accumulated.

Usage:

```
>>> trainingStat.reset()
```

update()

Updates the stats of the current session and resets the measures for next session.

Usage:

```
>>> trainingStat.update()
```

class learningStats.learningStats

This class provides mechanism to collect learning stats for training and testing, and displaying them efficiently.

Usage:

```
stats = learningStats()
```

(continues on next page)

(continued from previous page)

```

for epoch in range(100):
    tSt = datetime.now()

    stats.training.reset()
    for i in trainingLoop:
        # other main stuffs
        stats.training.correctSamples += numberOfCorrectClassification
        stats.training.numSamples    += numberOfSamplesProcessed
        stats.training.lossSum       += currentLoss
        stats.print(epoch, i, (datetime.now() - tSt).total_seconds())
    stats.training.update()

    stats.testing.reset()
    for i in testingLoop:
        # other main stuffs
        stats.testing.correctSamples += numberOfCorrectClassification
        stats.testing.numSamples    += numberOfSamplesProcessed
        stats.testing.lossSum       += currentLoss
        stats.print(epoch, i)
    stats.training.update()

```

plot (*figures=(1, 2), saveFig=False, path=""*)

Plots the available learning statistics.

Arguments:

- *figures*: Index of figure ID to plot on. Default is figure(1) for loss plot and figure(2) for accuracy plot.
- *saveFig* (``bool`): flag to save figure into a file.
- *path*: path to save the file. Default is `.`.

Usage:

```

# plot stats
stats.plot()

# plot stats figures specified
stats.print(figures=(10, 11))

```

print (*epoch, iter=None, timeElapsed=None*)

Prints the available learning statistics from the current session on the console. For Linux systems, prints the data on same terminal space (might not work properly on other systems).

Arguments:

- *epoch*: epoch counter to display (required).
- *iter*: iteration counter to display (not required).
- *timeElapsed*: runtime information (not required).

Usage:

```

# prints stats with epoch index provided
stats.print(epoch)

# prints stats with epoch index and iteration index provided
stats.print(epoch, iter=i)

```

(continues on next page)

(continued from previous page)

```
# prints stats with epoch index, iteration index and time elapsed information_
↪provided
stats.print(epoch, iter=i, timeElapsed=time)
```

save (filename="")

Saves the learning statistics logs.

Arguments:

- filename: filename to save the logs. accuracy.txt and loss.txt will be appended

Usage:

```
# save stats
stats.save()

# save stats filename specified
stats.save(filename='Run101-0.001-') # Run101-0.001-accuracy.txt and Run101-0.
↪001-loss.txt
```

update ()

Updates the stats for training and testing and resets the measures for next session.

Usage:

```
>>> stats.update()
```


OPTIMIZER

class optimizer.**Nadam**(*params*, *lr*=0.001, *betas*=(0.9, 0.999), *eps*=1e-08, *weight_decay*=0, *amsgrad*=False)

Implements Nadam algorithm. (Modified Adam from [PyTorch](#))

It has been proposed in [Incorporating Nesterov Momentum into Adam](#).

Arguments:

- *params* (iterable): iterable of parameters to optimize or dicts defining parameter groups.
- *lr* (float, optional): learning rate (default: 1e-3).
- *betas* (Tuple[float, float], optional): coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999)).
- *eps* (float, optional): term added to the denominator to improve numerical stability (default: 1e-8).
- *weight_decay* (float, optional): weight decay (L2 penalty) (default: 0).
- *amsgrad* (boolean, optional): whether to use the AMSGrad variant of this algorithm from the paper [On the Convergence of Adam and Beyond](#) (default: False).

step (*closure*=None)

Performs a single optimization step.

Arguments:

- **closure** (callable, optional): A closure that reevaluates the model and returns the loss.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

8.1 Usage:

```
>>> import slayerSNN as snn
```

- The **spike-layer** module will be available as `snn.layer`.
- The **yaml-parameter** module will be available as `snn.params`.
- The **spike-loss** module will be available as `snn.loss`.
- The **spike-classifier** module will be available as `snn.predict`.
- The **spike-IO** module will be available as `snn.io`.

8.2 Example:

The SNN parameters are stored in a yaml file. The structure of the yaml file follows the same hierarchy as the C++ SLAYER framework (see Network Description)

```
1 import slayerSNN as snn
2 # other imports and definitions
3
4 class Network(torch.nn.Module):
5     def __init__(self, netParams, device=device):
6         super(Network, self).__init__()
7         # initialize slayer
8         slayer = snn.layer(netParams['neuron'], netParams['simulation'], device=device)
9         self.sl = slayer
10        # define network functions
11        self.conv1 = slayer.conv(2, 16, 5, padding=1)
12        self.conv2 = slayer.conv(16, 32, 3, padding=1)
13        self.conv3 = slayer.conv(32, 64, 3, padding=1)
14        self.pool1 = slayer.pool(2)
15        self.pool2 = slayer.pool(2)
16        self.fcl = slayer.dense((8, 8, 64), 10)
```

(continues on next page)

(continued from previous page)

```

17
18     def forward(self, spikeInput):
19         spikeLayer1 = self.sl.spike(self.conv1(self.sl.psp(spikeInput))) # 32, 32, 16
20         spikeLayer2 = self.sl.spike(self.pool1(self.sl.psp(spikeLayer1))) # 16, 16, 16
21         spikeLayer3 = self.sl.spike(self.conv2(self.sl.psp(spikeLayer2))) # 16, 16, 32
22         spikeLayer4 = self.sl.spike(self.pool2(self.sl.psp(spikeLayer3))) # 8, 8, 32
23         spikeLayer5 = self.sl.spike(self.conv3(self.sl.psp(spikeLayer4))) # 8, 8, 64
24         spikeOut = self.sl.spike(self.fcl(self.sl.psp(spikeLayer5))) # 10
25         return spikeOut
26
27     # network
28     net = Network(snn.params('path to yaml file'))
29
30     # cost function
31     error = snn.loss(netParams)
32
33     # dataloader not shown. input and target are assumed to be available
34     output = net.forward(input)
35     loss = error.numSpikes(output, target)

```

Important: It is assumed that all the tensors that are being processed are 5 dimensional. (Batch, Channels, Height, Width, Time) or NCHWT format. The user must make sure that an input of correct dimension is supplied.

*If the layer does not have spatial dimension, the neurons can be distributed along either Channel, Height or Width dimension where Channel * Height * Width is equal to number of neurons. It is recommended (for speed reasons) to define the neurons in Channels dimension and make Height and Width dimension one.*

PYTHON MODULE INDEX

I

learningStats, [17](#)

O

optimizer, [21](#)

S

slayer, [3](#)
slayerSNN, [1](#)
spikeClassifier, [7](#)
spikeFileIO, [11](#)
spikeLoss, [9](#)

A

accuracy () (*learningStats.learningStat method*), 17

C

conv () (*slayer.spikeLayer method*), 3

D

delay () (*slayer.spikeLayer method*), 4
 delayShift () (*slayer.spikeLayer method*), 4
 dense () (*slayer.spikeLayer method*), 5
 dropout () (*slayer.spikeLayer method*), 5

E

encode1DnumSpikes () (*in module spikeFileIO*), 11
 encode1Dspikes () (*in module spikeFileIO*), 11
 encode2Dspikes () (*in module spikeFileIO*), 12
 encode3Dspikes () (*in module spikeFileIO*), 12
 event (*class in spikeFileIO*), 12

G

getClass () (*spikeClassifier.spikeClassifier static method*), 7

L

learningStat (*class in learningStats*), 17
 learningStats (*class in learningStats*), 17
 learningStats (*module*), 17
 loss () (*learningStats.learningStat method*), 17

N

Nadam (*class in optimizer*), 21
 numSpikes () (*spikeLoss.spikeLoss method*), 9

O

optimizer (*module*), 21

P

plot () (*learningStats.learningStats method*), 18
 pool () (*slayer.spikeLayer method*), 5
 print () (*learningStats.learningStats method*), 18
 psp () (*slayer.spikeLayer method*), 6

pspLayer () (*slayer.spikeLayer method*), 6

R

read1DnumSpikes () (*in module spikeFileIO*), 13
 read1Dspikes () (*in module spikeFileIO*), 13
 read2Dspikes () (*in module spikeFileIO*), 14
 read3Dspikes () (*in module spikeFileIO*), 14
 reset () (*learningStats.learningStat method*), 17

S

save () (*learningStats.learningStats method*), 19
 showTD () (*in module spikeFileIO*), 14
 slayer (*module*), 3
 slayerSNN (*module*), 1
 spike () (*slayer.spikeLayer method*), 6
 spikeArrayToEvent () (*in module spikeFileIO*), 15
 spikeClassifier (*class in spikeClassifier*), 7
 spikeClassifier (*module*), 7
 spikeFileIO (*module*), 11
 spikeLayer (*class in slayer*), 3
 spikeLoss (*class in spikeLoss*), 9
 spikeLoss (*module*), 9
 spikeTime () (*spikeLoss.spikeLoss method*), 9
 step () (*optimizer.Nadam method*), 21

T

toSpikeArray () (*spikeFileIO.event method*), 13
 toSpikeTensor () (*spikeFileIO.event method*), 13

U

update () (*learningStats.learningStat method*), 17
 update () (*learningStats.learningStats method*), 19

Y

yamlParams (*class in slayer*), 6