CONTENTS:

1 Introduction 1
Python Module Index 225
Index 227
The brain is the perfect place to look for inspiration to develop more efficient neural networks. One of the main differences with modern deep learning is that the brain encodes information in spikes rather than continuous activations. snnTorch is a Python package for performing gradient-based learning with spiking neural networks. It extends the capabilities of PyTorch, taking advantage of its GPU accelerated tensor computation and applying it to networks of spiking neurons. Pre-designed spiking neuron models are seamlessly integrated within the PyTorch framework and can be treated as recurrent activation units.

If you like this project, please consider starring this repo as it is the easiest and best way to support it.

If you have issues, comments, or are looking for advice on training spiking neural networks, you can open an issue, a discussion, or chat in our discord channel.

1.1 snnTorch Structure

snnTorch contains the following components:
snnTorch is designed to be intuitively used with PyTorch, as though each spiking neuron were simply another activation in a sequence of layers. It is therefore agnostic to fully-connected layers, convolutional layers, residual connections, etc.

At present, the neuron models are represented by recursive functions which removes the need to store membrane potential traces for all neurons in a system in order to calculate the gradient. The lean requirements of snnTorch enable small and large networks to be viably trained on CPU, where needed. Provided that the network models and tensors are loaded onto CUDA, snnTorch takes advantage of GPU acceleration in the same way as PyTorch.

## 1.2 Citation

If you find snnTorch useful in your work, please cite the following source:


```latex
@article{eshraghian2021training,
  title = {Training spiking neural networks using lessons from deep learning},
  author = {Eshraghian, Jason K and Ward, Max and Neftci, Emre and Wang, Xinxin and Lenz, Gregor and Dwivedi, Girish and Bennamoun, Mohammed and Jeong, Doo Seok and Lu, Wei D},
  year = {2021}
}
```

Let us know if you are using snnTorch in any interesting work, research or blogs, as we would love to hear more about it! Reach out at snntorch@gmail.com.

## 1.3 Requirements

The following packages need to be installed to use snnTorch:

- torch >= 1.1.0
- numpy >= 1.17
- pandas
- matplotlib
- math
They are automatically installed if snnTorch is installed using the pip command. Ensure the correct version of torch is installed for your system to enable CUDA compatibility.

## 1.4 Installation

Run the following to install:

```
$ python
$ pip install snntorch
```

To install snnTorch from source instead:

```
$ git clone https://github.com/jeshraghian/snnTorch
$ cd snntorch
$ python setup.py install
```

To install snntorch with conda:

```
$ conda install -c conda-forge snntorch
```

To install for an Intelligent Processing Units (IPU) based build using Graphcore’s accelerators:

```
$ pip install snntorch-iper
```

## 1.5 API & Examples

A complete API is available [here](#). Examples, tutorials and Colab notebooks are provided.

## 1.6 Quickstart

Here are a few ways you can get started with snnTorch:

- Quickstart Notebook (Opens in Colab)
- The API Reference
- Examples
- Tutorials

For a quick example to run snnTorch, see the following snippet, or test the quickstart notebook:

```python
import torch, torch.nn as nn
import snntorch as snn
from snntorch import surrogate

num_steps = 25  # number of time steps
batch_size = 1
beta = 0.5  # neuron decay rate
spike_grad = surrogate.fast_sigmoid()
```

(continues on next page)
net = nn.Sequential(
    nn.Conv2d(1, 8, 5),
    nn.MaxPool2d(2),
    snn.Leaky(beta=beta, init_hidden=True, spike_grad=spike_grad),
    nn.Conv2d(8, 16, 5),
    nn.MaxPool2d(2),
    snn.Leaky(beta=beta, init_hidden=True, spike_grad=spike_grad),
    nn.Flatten(),
    nn.Linear(16 * 4 * 4, 10),
    snn.Leaky(beta=beta, init_hidden=True, spike_grad=spike_grad, output=True)
)

# random input data
data_in = torch.rand(num_steps, batch_size, 1, 28, 28)
spike_recording = []
for step in range(num_steps):
    spike, state = net(data_in[step])
    spike_recording.append(spike)

If you’re feeling lazy and want the training process to be taken care of:

import snntorch.functional as SF
from snntorch import backprop

# correct class should fire 80% of the time
loss_fn = SF.mse_count_loss(correct_rate=0.8, incorrect_rate=0.2)
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3, betas=(0.9, 0.999))

# train for one epoch using the backprop through time algorithm
# assume train_loader is a DataLoader with time-varying input
avg_loss = backprop.BPTT(net, train_loader, optimizer=optimizer,
                          num_steps=num_steps, criterion=loss_fn)

1.7 A Deep Dive into SNNs

If you wish to learn all the fundamentals of training spiking neural networks, from neuron models, to the neural code, up to backpropagation, the snnTorch tutorial series is a great place to begin. It consists of interactive notebooks with complete explanations that can get you up to speed.

<table>
<thead>
<tr>
<th>Tutorial</th>
<th>Title</th>
<th>Colab Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutorial 1</td>
<td>Spike Encoding with snnTorch</td>
<td></td>
</tr>
<tr>
<td>Tutorial 2</td>
<td>The Leaky Integrate and Fire Neuron</td>
<td></td>
</tr>
<tr>
<td>Tutorial 3</td>
<td>A Feedforward Spiking Neural Network</td>
<td></td>
</tr>
<tr>
<td>Tutorial 4</td>
<td>2nd Order Spiking Neuron Models (Optional)</td>
<td></td>
</tr>
<tr>
<td>Tutorial 5</td>
<td>Training Spiking Neural Networks with snnTorch</td>
<td></td>
</tr>
<tr>
<td>Tutorial 6</td>
<td>Surrogate Gradient Descent in a Convolutional SNN</td>
<td></td>
</tr>
<tr>
<td>Tutorial 7</td>
<td>Neuromorphic Datasets with Tonic + snnTorch</td>
<td></td>
</tr>
</tbody>
</table>
1.8 Intelligent Processing Unit (IPU) Acceleration

snnTorch has been optimized for Graphcore’s IPU accelerators. To install an IPU based build of snnTorch:

$ pip install snntorch-ipu

Low-level custom operations for IPU compatibility will be automatically compiled when import snntorch is called for the first time.

When updating the Poplar SDK, these operations may need to be recompiled. This can be done by reinstalling snntorch-ipu, or deleting files in the base directory with an .so extension.

The snntorch.backprop module, and several functions from snntorch.functional and snntorch.surrogate, are incompatible with IPUs, but can be recreated using PyTorch primitives.

Additional requirements include:

• poptorch
• The Poplar SDK

Refer to Graphcore’s documentation for installation instructions of poptorch and the Poplar SDK.

The homepage for the snnTorch IPU project can be found here. A tutorial for training SNNs is provided here.

1.9 Contributing

If you’re ready to contribute to snnTorch, instructions to do so can be found here.

1.10 Acknowledgments

snnTorch is currently maintained by the UCSC Neuromorphic Computing Group. It was initially developed by Jason K. Eshraghian in the Lu Group (University of Michigan).

Additional contributions were made by Vincent Sun, Peng Zhou, Ridger Zhu, Xinxin Wang, and Emre Neftci.

1.11 License & Copyright

snnTorch source code is published under the terms of the MIT License. snnTorch’s documentation is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License (CC BY-SA 3.0).
The brain is the perfect place to look for inspiration to develop more efficient neural networks. One of the main differences with modern deep learning is that the brain encodes information in spikes rather than continuous activations. snnTorch is a Python package for performing gradient-based learning with spiking neural networks. It extends the capabilities of PyTorch, taking advantage of its GPU accelerated tensor computation and applying it to networks of spiking neurons. Pre-designed spiking neuron models are seamlessly integrated within the PyTorch framework and can be treated as recurrent activation units.

If you like this project, please consider starring this repo as it is the easiest and best way to support it.

If you have issues, comments, or are looking for advice on training spiking neural networks, you can open an issue, a discussion, or chat in our discord channel.

**snnTorch Structure**

snnTorch contains the following components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>snntorch</td>
<td>a spiking neuron library like torch.nn, deeply integrated with autograd</td>
</tr>
<tr>
<td>snntorch.backprop</td>
<td>variations of backpropagation commonly used with SNNs</td>
</tr>
<tr>
<td>snntorch.functional</td>
<td>common arithmetic operations on spikes, e.g., loss, regularization etc.</td>
</tr>
<tr>
<td>snntorch.spikegen</td>
<td>a library for spike generation and data conversion</td>
</tr>
<tr>
<td>snntorch.spikeplot</td>
<td>visualization tools for spike-based data using matplotlib and celluloid</td>
</tr>
<tr>
<td>snntorch.spikevision</td>
<td>contains popular neuromorphic datasets</td>
</tr>
<tr>
<td>snntorch.surrogate</td>
<td>optional surrogate gradient functions</td>
</tr>
<tr>
<td>snntorch.utils</td>
<td>dataset utility functions</td>
</tr>
</tbody>
</table>

snnTorch is designed to be intuitively used with PyTorch, as though each spiking neuron were simply another activation in a sequence of layers. It is therefore agnostic to fully-connected layers, convolutional layers, residual connections, etc.

At present, the neuron models are represented by recursive functions which removes the need to store membrane potential traces for all neurons in a system in order to calculate the gradient. The lean requirements of snnTorch enable
small and large networks to be viably trained on CPU, where needed. Provided that the network models and tensors are loaded onto CUDA, snnTorch takes advantage of GPU acceleration in the same way as PyTorch.

**Citation**

If you find snnTorch useful in your work, please cite the following source:


```latex
@article{eshraghian2021training,
  title = {Training spiking neural networks using lessons from deep learning},
  author = {Eshraghian, Jason K and Ward, Max and Neftci, Emre and Wang, Xinxin and Lenz, Gregor and Dwivedi, Girish and Bennamoun, Mohammed and Jeong, Doo Seok and Lu, Wei D},
  year = {2021}
}
```

Let us know if you are using snnTorch in any interesting work, research or blogs, as we would love to hear more about it! Reach out at snntorch@gmail.com.

**Requirements**

The following packages need to be installed to use snnTorch:

- `torch >= 1.1.0`
- `numpy >= 1.17`
- `pandas`
- `matplotlib`
- `math`

They are automatically installed if snnTorch is installed using the pip command. Ensure the correct version of torch is installed for your system to enable CUDA compatibility.

**Installation**

Run the following to install:

```
$ python
$ pip install snntorch
```

To install snnTorch from source instead:

```
$ git clone https://github.com/jeshraghian/snnTorch
$ cd snntorch
$ python setup.py install
```

To install snntorch with conda:
$ conda install -c conda-forge snntorch

To install for an Intelligent Processing Units (IPU) based build using Graphcore’s accelerators:

$ pip install snntorch-ipu

### API & Examples

A complete API is available [here](#). Examples, tutorials and Colab notebooks are provided.

### Quickstart

Here are a few ways you can get started with snnTorch:

- Quickstart Notebook (Opens in Colab)
- The API Reference
- Examples
- Tutorials

For a quick example to run snnTorch, see the following snippet, or test the quickstart notebook:

```python
import torch, torch.nn as nn
import snntorch as snn
from snntorch import surrogate

num_steps = 25  # number of time steps
batch_size = 1
beta = 0.5  # neuron decay rate
spike_grad = surrogate.fast_sigmoid()

net = nn.Sequential(
    nn.Conv2d(1, 8, 5),
    nn.MaxPool2d(2),
    snn.Leaky(beta=beta, init_hidden=True, spike_grad=spike_grad),
    nn.Conv2d(8, 16, 5),
    nn.MaxPool2d(2),
    snn.Leaky(beta=beta, init_hidden=True, spike_grad=spike_grad),
    nn.Flatten(),
    nn.Linear(16 * 4 * 4, 10),
    snn.Leaky(beta=beta, init_hidden=True, spike_grad=spike_grad, output=True)
)

# random input data
data_in = torch.rand(num_steps, batch_size, 1, 28, 28)

spike_recording = []

for step in range(num_steps):
    spike, state = net(data_in[step])
    spike_recording.append(spike)
```
If you're feeling lazy and want the training process to be taken care of:

```python
import snntorch.functional as SF
from snntorch import backprop

# correct class should fire 80% of the time
loss_fn = SF.mse_count_loss(correct_rate=0.8, incorrect_rate=0.2)
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3, betas=(0.9, 0.999))

# train for one epoch using the backprop through time algorithm
# assume train_loader is a DataLoader with time-varying input
avg_loss = backprop.BPTT(net, train_loader, optimizer=optimizer,
                          num_steps=num_steps, criterion=loss_fn)
```

### A Deep Dive into SNNs

If you wish to learn all the fundamentals of training spiking neural networks, from neuron models, to the neural code, up to backpropagation, the snnTorch tutorial series is a great place to begin. It consists of interactive notebooks with complete explanations that can get you up to speed.

<table>
<thead>
<tr>
<th>Tutorial</th>
<th>Title</th>
<th>Colab Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutorial 1</td>
<td>Spike Encoding with snnTorch</td>
<td></td>
</tr>
<tr>
<td>Tutorial 2</td>
<td>The Leaky Integrate and Fire Neuron</td>
<td></td>
</tr>
<tr>
<td>Tutorial 3</td>
<td>A Feedforward Spiking Neural Network</td>
<td></td>
</tr>
<tr>
<td>Tutorial 4</td>
<td>2nd Order Spiking Neuron Models (Optional)</td>
<td></td>
</tr>
<tr>
<td>Tutorial 5</td>
<td>Training Spiking Neural Networks with snnTorch</td>
<td></td>
</tr>
<tr>
<td>Tutorial 6</td>
<td>Surrogate Gradient Descent in a Convolutional SNN</td>
<td></td>
</tr>
<tr>
<td>Tutorial 7</td>
<td>Neuromorphic Datasets with Tonic + snnTorch</td>
<td></td>
</tr>
</tbody>
</table>

### Advanced Tutorials

<table>
<thead>
<tr>
<th>Title</th>
<th>Colab Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Coding</td>
<td></td>
</tr>
<tr>
<td>Accelerating snnTorch on IPUs</td>
<td></td>
</tr>
</tbody>
</table>

### Intelligent Processing Unit (IPU) Acceleration

snnTorch has been optimized for Graphcore’s IPU accelerators. To install an IPU based build of snnTorch:

```
$ pip install snntorch-ipu
```

Low-level custom operations for IPU compatibility will be automatically compiled when `import snntorch` is called for the first time.

When updating the Poplar SDK, these operations may need to be recompiled. This can be done by reinstalling `snntorch-ipu`, or deleting files in the base directory with an `.so` extension.

The `snntorch.backprop` module, and several functions from `snntorch.functional` and `snntorch.surrogate`, are incompatible with IPUs, but can be recreated using PyTorch primitives.

Additional requirements include:

- poptorch
- The Poplar SDK
Refer to Graphcore’s documentation for installation instructions of poptorch and the Poplar SDK.
The homepage for the snnTorch IPU project can be found here. A tutorial for training SNNs is provided here.

**Contributing**

If you’re ready to contribute to snnTorch, instructions to do so can be found here.

**Acknowledgments**

snnTorch is currently maintained by the UCSC Neuromorphic Computing Group. It was initially developed by Jason K. Eshraghian in the Lu Group (University of Michigan).
Additional contributions were made by Vincent Sun, Peng Zhou, Ridger Zhu, Xinxin Wang, and Emre Neftci.

**License & Copyright**

snnTorch source code is published under the terms of the MIT License. snnTorch’s documentation is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License (CC BY-SA 3.0).

### 1.11.2 Installation

**Stable release**

To install snntorch, run this command in your terminal:

```bash
$ pip install snntorch
```

This is the preferred method to install snntorch, as it will always install the most recent stable release.

If you don’t have pip installed, this Python installation guide can guide you through the process.

To install snntorch with conda:

```bash
$ conda install -c conda-forge snntorch
```

**From sources**

The sources for snntorch can be downloaded from the Github repo.

You can either clone the public repository:

```bash
$ git clone git://github.com/jeshraghian/snntorch
```

Or download the tarball:

```bash
$ curl -OJL https://github.com/jeshraghian/snntorch/tarball/master
```

Once you have a copy of the source, you can install it with:

```bash
$ python setup.py install
```
1.11.3 snntorch

snnTorch Neurons

`snntorch` is designed to be intuitively used with PyTorch, as though each spiking neuron were simply another activation in a sequence of layers.

A variety of spiking neuron classes are available which can simply be treated as activation units with PyTorch. Each layer of spiking neurons are therefore agnostic to fully-connected layers, convolutional layers, residual connections, etc.

The neuron models are represented by recursive functions which removes the need to store membrane potential traces in order to calculate the gradient. The lean requirements of `snntorch` enable small and large networks to be viably trained on CPU, where needed. Being deeply integrated with `torch.autograd`, `snntorch` is able to take advantage of GPU acceleration in the same way as PyTorch.

By default, PyTorch’s autodifferentiation mechanism in `torch.autograd` nulls the gradient signal of the spiking neuron graph due to non-differentiable spiking threshold functions. `snntorch` overrides the default gradient by using `snntorch.neurons.Heaviside`. Alternative options exist in `snntorch.surrogate`.

At present, the neurons available in `snntorch` are variants of the Leaky Integrate-and-Fire neuron model:

- **Leaky** - 1st-Order Leaky Integrate-and-Fire Neuron
- **RLeaky** - As above, with recurrent connections for output spikes
- **Synaptic** - 2nd-Order Integrate-and-Fire Neuron (including synaptic conductance)
- **RSynaptic** - As above, with recurrent connections for output spikes
- **Lapicque** - Lapicque’s RC Neuron Model
- **Alpha** - Alpha Membrane Model

Additional models include spiking-LSTMs and spiking-ConvLSTMs:

- **SLSTM** - Spiking long short-term memory cell with state-thresholding
- **SConv2dLSTM** - Spiking 2d convolutional short-term memory cell with state thresholding

**How to use snnTorch’s neuron models**

The following arguments are common across all neuron models:

- **threshold** - firing threshold of the neuron
- **spike_grad** - surrogate gradient function (see `snntorch.surrogate`)
- **init_hidden** - setting to True hides all neuron states as instance variables to reduce code complexity
- **inhibition** - setting to True enables only the neuron with the highest membrane potential to fire in a dense layer (not for use in convs etc.)
- **learn_beta** - setting to True enables the decay rate to be a learnable parameter
- **learn_threshold** - setting to True enables the threshold to be a learnable parameter
- **reset_mechanism** - options include `subtract` (reset-by-subtraction), `zero` (reset-to-zero), and `none` (no reset mechanism: i.e., leaky integrator neuron)
- **output** - if `init_hidden=True`, the spiking neuron will only return the output spikes. Setting `output=True` enables the hidden state(s) to be returned as well. Useful when using `torch.nn.sequential`.

Leaky integrate-and-fire neuron models also include:
• **beta** - decay rate of membrane potential, clipped between 0 and 1 during the forward-pass. Can be a single-value tensor (same decay for all neurons in a layer), or can be multi-valued (individual weights p/neuron in a layer). More complex neurons include additional parameters, such as **alpha**.

Recurrent spiking neuron models, such as `snntorch.RLeaky` and `snntorch.RSynaptic` explicitly pass the output spike back to the input. Such neurons include additional arguments:

• **V** - Recurrent weight. Can be a single-valued tensor (same weight across all neurons in a layer), or multi-valued tensor (individual weights p/neuron in a layer).

• **learn_V** - defaults to True, which enables V to be a learnable parameter.

Spiking neural networks can be constructed using a combination of the `snntorch` and `torch.nn` packages.

Example:

```python
import torch
import torch.nn as nn
import snntorch as snn

alpha = 0.9
beta = 0.85
num_steps = 100

# Define Network
class Net(nn.Module):
    def __init__(self):
        super().__init__()

        # initialize layers
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.Leaky(beta=beta)
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        self.lif2 = snn.Leaky(beta=beta)

    def forward(self, x):
        mem1 = self.lif1.init_leaky()
        mem2 = self.lif2.init_leaky()

        spk2_rec = []  # Record the output trace of spikes
        mem2_rec = []  # Record the output trace of membrane potential

        for step in range(num_steps):
            cur1 = self.fc1(x.flatten(1))
            spk1, mem1 = self.lif1(cur1, mem1)
            cur2 = self.fc2(spk1)
            spk2, mem2 = self.lif2(cur2, mem2)

            spk2_rec.append(spk2)
            mem2_rec.append(mem2)

        return torch.stack(spk2_rec), torch.stack(mem2_rec)

net = Net().to(device)
```

(continues on next page)
In the above example, the hidden state `mem` must be manually initialized for each layer. This can be overcome by automatically instantiating neuron hidden states by invoking `init_hidden=True`.

In some cases (e.g., truncated backprop through time), it might be necessary to perform backward passes before all time steps have completed processing. This requires moving the time step for-loop out of the network and into the training-loop.

An example of this is shown below:

```python
import torch
import torch.nn as nn
import snntorch as snn

num_steps = 100

lif1 = snn.Leaky(beta=0.9, init_hidden=True) # only returns spk
lif2 = snn.Leaky(beta=0.9, init_hidden=True, output=True) # returns mem and spk if output=True

# Initialize Network
net = nn.Sequential(nn.Flatten(),
                    nn.Linear(784, 1000),
                    lif1,
                    nn.Linear(1000, 10),
                    lif2).to(device)

for step in range(num_steps):
    spk_out, mem_out = net(data)
```

Setting the hidden states to instance variables is necessary for calling the backpropagation methods available in `snntorch.backprop`, or for calling `nn.Sequential` from PyTorch.

Whenever a neuron is instantiated, it is added as a list item to the class variable `LIF.instances`. This helps the functions in `snntorch.backprop` keep track of what neurons are being used in the network, and when they must be detached from the computation graph.

In the above examples, the decay rate of membrane potential `beta` is treated as a hyperparameter. But it can also be configured as a learnable parameter, as shown below:

```python
import torch
import torch.nn as nn
import snntorch as snn

num_steps = 100

lif1 = snn.Leaky(beta=0.9, learn_beta=True, init_hidden=True) # only returns spk
lif2 = snn.Leaky(beta=0.5, learn_beta=True, init_hidden=True, output=True) # returns mem and spk if output=True
```
# Initialize Network

```python
net = nn.Sequential(nn.Flatten(),
    nn.Linear(784, 1000),
    lif1,
    nn.Linear(1000, 10),
    lif2).to(device)

for step in range(num_steps):
    spk_out, mem_out = net(data.view(batch_size, -1))
```

Here, $\beta$ is initialized to 0.9 for the first layer, and 0.5 for the second layer. Each layer then treats it as a learnable parameter, just like all the other network weights. In the event you wish to have a learnable decay rate for each neuron rather than each layer, the following example shows how:

```python
import torch
import torch.nn as nn
import snntorch as snn

num_steps = 100
num_hidden = 1000
num_output = 10

beta1 = torch.rand(num_hidden)  # randomly initialize beta as a vector
beta2 = torch.rand(num_output)

lif1 = snn.Leaky(beta=beta1, learn_beta=True, init_hidden=True)  # only returns spk
lif2 = snn.Leaky(beta=beta2, learn_beta=True, init_hidden=True, output=True)  # returns mem and spk if output=True

# Initialize Network
net = nn.Sequential(nn.Flatten(),
    nn.Linear(784, num_hidden),
    lif1,
    nn.Linear(1000, num_output),
    lif2).to(device)

for step in range(num_steps):
    spk_out, mem_out = net(data.view(batch_size, -1))
```

The same approach as above can be used for implementing learnable thresholds, using `learn_threshold=True`.

Each neuron has the option to inhibit other neurons within the same dense layer from firing. This can be invoked by setting `inhibition=True` when instantiating the neuron layer. It has not yet been implemented for networks other than fully-connected layers, so use with caution.
Neuron List

snn.Alpha

class snntorch._neurons.alpha.Alpha(alpha, beta, threshold=1.0, spike_grad=None, init_hidden=False, inhibition=False, learn_alpha=False, learn_beta=False, learn_threshold=False, reset_mechanism='zero', state_quant=False, output=False)

Bases: LIF

A variant of the leaky integrate and fire neuron where membrane potential follows an alpha function. The time course of the membrane potential response depends on a combination of exponentials. In general, this causes the change in membrane potential to experience a delay with respect to an input spike. For $U[T] > U_{th} S[T+1] = 1$.

**Warning:** For a positive input current to induce a positive membrane response, ensure $>.$

If $reset\_mechanism = \text{“zero”}$, then $I_{\text{exc}}, I_{\text{inh}}$ will both be set to 0 whenever the neuron emits a spike:

\[
\begin{align*}
I_{\text{exc}}[t+1] &= (I_{\text{exc}}[t] + I_{\text{in}}[t+1]) - R(I_{\text{exc}}[t] + I_{\text{in}}[t+1]) \\
I_{\text{inh}}[t+1] &= (I_{\text{inh}}[t] - I_{\text{in}}[t+1]) - R(I_{\text{inh}}[t] - I_{\text{in}}[t+1]) \\
U[t+1] &= (I_{\text{exc}}[t+1] + I_{\text{inh}}[t+1])
\end{align*}
\]

- $I_{\text{exc}}$ - Excitatory current
- $I_{\text{inh}}$ - Inhibitory current
- $I_{\text{in}}$ - Input current
- $U$ - Membrane potential
- $U_{th}$ - Membrane threshold
- $R$ - Reset mechanism, $R = 1$ if spike occurs, otherwise $R = 0$
- - Excitatory current decay rate
- - Inhibitory current decay rate
- $\log() - \log() + 1$

Example:

```python
import torch
import torch.nn as nn
import snntorch as snn

alpha = 0.9
beta = 0.8

# Define Network
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        # initialize layers
        self.fc1 = nn.Linear(num_inputs, num_hidden)
```

(continues on next page)
```python
def forward(self, x, syn_exc1, syn_inh1, mem1, spk1, syn_exc2, syn_inh2, mem2):
    cur1 = self.fc1(x)
    spk1, syn_exc1, syn_inh1, mem1 = self.lif1(cur1, syn_exc1, syn_inh1, mem1)
    cur2 = self.fc2(spk1)
    spk2, syn_exc2, syn_inh2, mem2 = self.lif2(cur2, syn_exc2, syn_inh2, mem2)
    return syn_exc1, syn_inh1, mem1, spk1, syn_exc2, syn_inh2, mem2, spk2
```

# Too many state variables which becomes cumbersome, so the following is also an option:

```python
alpha = 0.9
beta = 0.8
```

```python
net = nn.Sequential(nn.Linear(num_inputs, num_hidden),
                    snn.Alpha(alpha=alpha, beta=beta, init_hidden=True),
                    nn.Linear(num_hidden, num_outputs),
                    snn.Alpha(alpha=alpha, beta=beta, init_hidden=True, output=True))
```

classmethod detach_hidden()

Used to detach hidden states from the current graph. Intended for use in truncated backpropagation through time where hidden state variables are instance variables.

forward(input_, syn_exc=False, syn_inh=False, mem=False)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

classmethod reset_hidden()

Used to clear hidden state variables to zero. Intended for use where hidden state variables are instance variables.

training:  bool
snn.Lapicque

class snntorch._neurons.lapicque.Lapicque(beta=False, R=False, C=False, time_step=1, threshold=1.0, spike_grad=None, init_hidden=False, inhibition=False, learn_beta=False, learn_threshold=False, reset_mechanism='subtract', state_quant=False, output=False)

Bases: LIF

An extension of Lapicque’s experimental comparison between extracellular nerve fibers and an RC circuit. It is qualitatively equivalent to Leaky but defined using RC circuit parameters. Input stimulus is integrated by membrane potential which decays exponentially with a rate of beta. For $U[T] > U_{th}S[T+1] = 1$.

If reset_mechanism = “subtract”, then $U[t+1]$ will have threshold subtracted from it whenever the neuron emits a spike:

$$U[t+1] = I_{in}[t+1](\frac{T}{C}) + (1 - \frac{T}{\tau})U[t] - RU_{th}$$

If reset_mechanism = “zero”, then $U[t+1]$ will be set to 0 whenever the neuron emits a spike:

$$U[t+1] = I_{in}[t+1](\frac{T}{C}) + (1 - \frac{T}{\tau})U[t] - R(I_{in}[t+1](\frac{T}{C}) + (1 - \frac{T}{\tau})U[t])$$

- $I_{in}$ - Input current
- $U$ - Membrane potential
- $U_{th}$ - Membrane threshold
- $T$ - duration of each time step
- $R$ - Reset mechanism: if active, $R = 1$, otherwise $R = 0$
- $\tau$ - Membrane potential decay rate

Alternatively, the membrane potential decay rate can be specified instead:

$$\tau = e^{-1/RC}$$

- $\tau$ - Membrane potential decay rate
- $R$ - Parallel resistance of passive membrane (note: distinct from the reset $R$)
- $C$ - Parallel capacitance of passive membrane
- If only $R$ is defined, then $R$ will default to 1, and $C$ will be inferred.
- If $RC$ is defined, will be automatically calculated.
- If $R$ (and $R$) or (and $C$) are defined, the missing variable will be automatically calculated.
- Note that, $R$ and $C$ are treated as hard-wired physically plausible parameters, and are therefore not learnable. For a single-state neuron with a learnable decay rate, use `snn.Leaky` instead.

Example:

```python
import torch
import torch.nn as nn
import snntorch as snn

beta = 0.5
```

(continues on next page)
R = 1
C = 1.44

# Define Network
class Net(nn.Module):
  def __init__(self):
    super().__init__()
    # initialize layers
    self.fc1 = nn.Linear(num_inputs, num_hidden)
    self.lif1 = snn.Lapicque(beta=beta)
    self.fc2 = nn.Linear(num_hidden, num_outputs)
    self.lif2 = snn.Lapicque(R=R, C=C)  # lif1 and lif2 are approximately
equivalent
  def forward(self, x, mem1, spk1, mem2):
    cur1 = self.fc1(x)
    spk1, mem1 = self.lif1(cur1, mem1)
    cur2 = self.fc2(spk1)
    spk2, mem2 = self.lif2(cur2, mem2)
    return mem1, spk1, mem2, spk2

For further reading, see:


Although Lapicque did not formally introduce this as an integrate-and-fire neuron model, we pay homage to his discovery of an RC circuit mimicking the dynamics of synaptic current.

Parameters

- **beta** (float or torch.tensor, Optional) – RC potential decay rate. Clipped between 0 and 1 during the forward-pass. May be a single-valued tensor (i.e., equal decay rate for all neurons in a layer), or multi-valued (one weight per neuron).
- **R** (int or torch.tensor, Optional) – Resistance of RC circuit
- **C** (int or torch.tensor, Optional) – Capacitance of RC circuit
- **time_step** (float, Optional) – time step precision. Defaults to 1
- **threshold** (float, optional) – Threshold for mem to reach in order to generate a spike S=1. Defaults to 1
- **spike_grad** (surrogate gradient function from snntorch.surrogate, optional) – Surrogate gradient for the term dS/dU. Defaults to None (corresponds to Heaviside surrogate gradient. See snntorch.surrogate for more options)
- **init_hidden** (bool, optional) – Instantiates state variables as instance variables. Defaults to False
- **inhibition** (bool, optional) – If True, suppresses all spiking other than the neuron with the highest state. Defaults to False

Chapter 1. Introduction
• **learn_beta** *(bool, optional)* – Option to enable learnable beta. Defaults to False

• **learn_threshold** *(bool, optional)* – Option to enable learnable threshold. Defaults to False

• **reset_mechanism** *(str, optional)* – Defines the reset mechanism applied to \( m \) each time the threshold is met. Reset-by-subtraction: “subtract”, reset-to-zero: “zero”, none: “none”. Defaults to “none”

• **state_quant** *(quantization function from snntorch.quant, optional)* – If specified, hidden state \( m \) is quantized to a valid state for the forward pass. Defaults to False

• **output** *(bool, optional)* – If True as well as \( init\_hidden=True \), states are returned when neuron is called. Defaults to False

**Inputs:** input_, mem_0

• **input** of shape *(batch, input_size)*: tensor containing input features

• **mem_0** of shape *(batch, input_size)*: tensor containing the initial membrane potential for each element in the batch.

**Outputs:** spk, mem_1

• **spk** of shape *(batch, input_size)*: tensor containing the output spikes.

• **mem_1** of shape *(batch, input_size)*: tensor containing the next membrane potential for each element in the batch

**Learnable Parameters:**

• **Lapcique.beta** *(torch.Tensor)* - optional learnable weights must be manually passed in, of shape 1 or *(input_size)*.

• **Lapcique.threshold** *(torch.Tensor)* - optional learnable thresholds must be manually passed in, of shape 1 or *(input_size)*.

**classmethod detach_hidden()**

Returns the hidden states, detached from the current graph. Intended for use in truncated backpropagation through time where hidden state variables are instance variables.

**forward**(input_, mem=False)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**classmethod reset_hidden()**

Used to clear hidden state variables to zero. Intended for use where hidden state variables are instance variables.

**training:** bool
snn.Leaky

class snntorch._neurons.leaky.Leaky(beta, threshold=1.0, spike_grad=None, init_hidden=False, inhibition=False, learn_beta=False, learn_threshold=False, reset_mechanism='subtract', state_quant=False, output=False)

Bases: LIF

First-order leaky integrate-and-fire neuron model. Input is assumed to be a current injection. Membrane potential decays exponentially with rate beta. For $U[T] > U_{thr}, S[T + 1] = 1$.

If reset_mechanism = “subtract”, then $U[t + 1]$ will have threshold subtracted from it whenever the neuron emits a spike:

$$U[t + 1] = U[t] + I_{in}[t + 1] - RU_{thr}$$

If reset_mechanism = “zero”, then $U[t + 1]$ will be set to 0 whenever the neuron emits a spike:

$$U[t + 1] = U[t] + I_{syn}[t + 1] - R(U[t] + I_{in}[t + 1])$$

- $I_{in}$ - Input current
- $U$ - Membrane potential
- $U_{thr}$ - Membrane threshold
- $R$ - Reset mechanism: if active, $R = 1$, otherwise $R = 0$
- - Membrane potential decay rate

Example:

```python
import torch
import torch.nn as nn
import snntorch as snn

beta = 0.5

# Define Network
class Net(nn.Module):
    def __init__(self):
        super().__init__()

        # initialize layers
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.Leaky(beta=beta)
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        self.lif2 = snn.Leaky(beta=beta)

    def forward(self, x, mem1, spk1, mem2):
        cur1 = self.fc1(x)
        spk1, mem1 = self.lif1(cur1, mem1)
        cur2 = self.fc2(spk1)
        spk2, mem2 = self.lif2(cur2, mem2)
        return mem1, spk1, mem2, spk2
```

Parameters
• **beta** (*float or torch.tensor*) – membrane potential decay rate. Clipped between 0 and 1 during the forward-pass. May be a single-valued tensor (i.e., equal decay rate for all neurons in a layer), or multi-valued (one weight per neuron).

• **threshold** (*float, optional*) – Threshold for mem to reach in order to generate a spike $S=1$. Defaults to 1

• **spike_grad** (*surrogate gradient function from snntorch.surrogate, optional*) – Surrogate gradient for the term $dS/dU$. Defaults to None (corresponds to Heaviside surrogate gradient. See snntorch.surrogate for more options)

• **init_hidden** (*bool, optional*) – Instantiates state variables as instance variables. Defaults to False

• **inhibition** (*bool, optional*) – If True, suppresses all spiking other than the neuron with the highest state. Defaults to False

• **learn_beta** (*bool, optional*) – Option to enable learnable beta. Defaults to False

• **learn_threshold** (*bool, optional*) – Option to enable learnable threshold. Defaults to False

• **reset_mechanism** (*str, optional*) – Defines the reset mechanism applied to mem each time the threshold is met. Reset-by-subtraction: “subtract”, reset-to-zero: “zero, none: “none”. Defaults to “subtract”

• **state_quant** (*quantization function from snntorch.quant, optional*) – If specified, hidden state mem is quantized to a valid state for the forward pass. Defaults to False

• **output** (*bool, optional*) – If True as well as init_hidden=True, states are returned when neuron is called. Defaults to False

**Inputs:** input_, mem_0

- **input_** of shape *(batch, input_size)*: tensor containing input features
- **mem_0** of shape *(batch, input_size)*: tensor containing the initial membrane potential for each element in the batch.

**Outputs:** spk, syn_1, mem_1

- **spk** of shape *(batch, input_size)*: tensor containing the output spikes.
- **mem_1** of shape *(batch, input_size)*: tensor containing the next membrane potential for each element in the batch

**Learnable Parameters:**

- **Leaky.beta** *(torch.Tensor)* - optional learnable weights must be manually passed in, of shape $I$ or $(input\_size)$.
- **Leaky.threshold** *(torch.Tensor)* - optional learnable thresholds must be manually passed in, of shape $I$ or $(input\_size)$.

**classmethod detach_hidden()**

Returns the hidden states, detached from the current graph. Intended for use in truncated backpropagation through time where hidden state variables are instance variables.

**forward(input_, mem=False)**

Defines the computation performed at every call.

Should be overridden by all subclasses.
classmethod reset_hidden()

Used to clear hidden state variables to zero. Intended for use where hidden state variables are instance variables. Assumes hidden states have a batch dimension already.

training: bool

snn.RLeaky

class snntorch._neurons.rleaky.RLeaky(beta, V=1.0, all_to_all=True, linear_features=None, conv2d_channels=None, kernel_size=None, threshold=1.0, spike_grad=None, init_hidden=False, inhibition=False, learn_beta=False, learn_threshold=False, learn_recurrent=True, reset_mechanism='subtract', state_quant=False, output=False)

Bases: LIF

First-order recurrent leaky integrate-and-fire neuron model. Input is assumed to be a current injection appended to the voltage spike output. Membrane potential decays exponentially with rate beta. For $U[T] > U_{thr}$, $S[T+1] = 1$.

If reset_mechanism = “subtract”, then $U[t+1]$ will have threshold subtracted from it whenever the neuron emits a spike:

$$U[t+1] = U[t] + I_{in}[t+1] + V(S_{out}[t]) - R U_{thr}$$

Where $V(\cdot)$ acts either as a linear layer, a convolutional operator, or elementwise product on $S_{out}$.

- If all_to_all = “True” and linear_features is specified, then $V(\cdot)$ acts as a recurrent linear layer of the same size as $S_{out}$.
- If all_to_all = “True” and conv2d_channels and kernel_size are specified, then $V(\cdot)$ acts as a recurrent convolutional layer with padding to ensure the output matches the size of the input.
- If all_to_all = “False”, then $V(\cdot)$ acts as an elementwise multiplier with $V$.

If reset_mechanism = “zero”, then $U[t+1]$ will be set to 0 whenever the neuron emits a spike:

$$U[t+1] = U[t] + I_{in}[t+1] + V(S_{out}[t]) - R(U[t] + I_{in}[t+1] + V(S_{out}[t])$$

- $I_{in}$ - Input current
- $U$ - Membrane potential
- $U_{thr}$ - Membrane threshold
- $S_{out}$ - Output spike
- $R$ - Reset mechanism: if active, $R = 1$, otherwise $R = 0$
- $V$ - Explicit recurrent weight

Example:
import torch
import torch.nn as nn
import snntorch as snn

beta = 0.5

# shared recurrent connection for a given layer
V1 = 0.5

# independent connection per neuron
V2 = torch.rand(num_outputs)

# Define Network
class Net(nn.Module):
    def __init__(self):
        super().__init__()

        # initialize layers
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.RLeaky(beta=beta, linear_features=num_hidden)
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        self.lif2 = snn.RLeaky(beta=beta, all_to_all=False, V=V1)

    def forward(self, x, mem1, spk1, mem2):
        cur1 = self.fc1(x)
        spk1, mem1 = self.lif1(cur1, spk1, mem1)
        cur2 = self.fc2(spk1)
        spk2, mem2 = self.lif2(cur2, spk2, mem2)

        return mem1, spk1, mem2, spk2

Parameters

- **beta** *(float or torch.tensor)* – membrane potential decay rate. Clipped between 0 and 1 during the forward-pass. May be a single-valued tensor (i.e., equal decay rate for all neurons in a layer), or multi-valued (one weight per neuron).

- **V** *(float or torch.tensor)* – Recurrent weights to scale output spikes, only used when *all_to_all=False*. Defaults to 1.

- **all_to_all** *(bool, optional)* – Enables output spikes to be connected in dense or convolutional recurrent structures instead of 1-to-1 connections. Defaults to True.

- **linear_features** *(int, optional)* – Size of each output sample. Must be specified if *all_to_all=True* and the input data is 1D. Defaults to None

- **conv2d_channels** *(int, optional)* – Number of channels in each output sample. Must be specified if *all_to_all=True* and the input data is 3D. Defaults to None

- **kernel_size** *(int or tuple)* – Size of the convolving kernel. Must be specified if *all_to_all=True* and the input data is 3D. Defaults to None

- **threshold** *(float, optional)* – Threshold for *mem* to reach in order to generate a spike $S=1$. Defaults to 1

- **spike_grad** *(surrogate gradient function from snntorch.surrogate, optional)* – Surrogate gradient for the term $dS/dU$. Defaults to None (corresponds to Heaviside surrogate gradient. See snntorch.surrogate for more options)
• **init_hidden** *(bool, optional)* – Instantiates state variables as instance variables. Defaults to False

• **inhibition** *(bool, optional)* – If True, suppresses all spiking other than the neuron with the highest state. Defaults to False

• **learn_beta** *(bool, optional)* – Option to enable learnable beta. Defaults to False

• **learn_recurrent** *(bool, optional)* – Option to enable learnable recurrent weights. Defaults to True

• **learn_threshold** *(bool, optional)* – Option to enable learnable threshold. Defaults to False

• **reset_mechanism** *(str, optional)* – Defines the reset mechanism applied to mem each time the threshold is met. Reset-by-subtraction: “subtract”, reset-to-zero: “zero, none: “none”. Defaults to “subtract”

• **state_quant** *(quantization function from snntorch.quant, optional)* – If specified, hidden state mem is quantized to a valid state for the forward pass. Defaults to False

• **output** *(bool, optional)* – If True as well as init_hidden=True, states are returned when neuron is called. Defaults to False

**Inputs:** input_, spk_0, mem_0

• **input_** of shape *(batch, input_size)*: tensor containing input features

• **spk_0** of shape *(batch, input_size)*: tensor containing output spike features

• **mem_0** of shape *(batch, input_size)*: tensor containing the initial membrane potential for each element in the batch.

**Outputs:** spk_1, mem_1

• **spk_1** of shape *(batch, input_size)*: tensor containing the output spikes.

• **mem_1** of shape *(batch, input_size)*: tensor containing the next membrane potential for each element in the batch

**Learnable Parameters:**

• **RLeaky.beta** *(torch.Tensor)* - optional learnable weights must be manually passed in, of shape 1 or (input_size).

• **RLeaky.recurrent.weight** *(torch.Tensor)* - optional learnable weights are automatically generated if all_to_all=True. RLeaky.recurrent stores a nn.Linear or nn.Conv2d layer depending on input arguments provided.

• **RLeaky.V** *(torch.Tensor)* - optional learnable weights must be manually passed in, of shape 1 or (input_size). It is only used where all_to_all=False for 1-to-1 recurrent connections.

• **RLeaky.threshold** *(torch.Tensor)* - optional learnable thresholds must be manually passed in, of shape 1 or (input_size).

**classmethod detach_hidden()**

Returns the hidden states, detached from the current graph. Intended for use in truncated backpropagation through time where hidden state variables are instance variables.
forward(input_, spk=False, mem=False)
    Defines the computation performed at every call.
    Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

classmethod reset_hidden()
    Used to clear hidden state variables to zero. Intended for use where hidden state variables are instance variables. Assumes hidden states have a batch dimension already.

training: bool
class snntorch._neurons.rleaky.RecurrentOneToOne(V)
    Bases: Module
    forward(x)
        Defines the computation performed at every call.
        Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

snn.RSynaptic
class snntorch._neurons.rsynaptic.RSynaptic(alpha, beta, V=1.0, all_to_all=True, linear_features=None, conv2d_channels=None, kernel_size=None, threshold=1.0, spike_grad=None, init_hidden=False, inhibition=False, learn_alpha=False, learn_beta=False, learn_threshold=False, learn_recurrent=True, reset_mechanism='subtract', state_quant=False, output=False)
    Bases: LIF

2nd order recurrent leaky integrate and fire neuron model accounting for synaptic conductance. The synaptic current jumps upon spike arrival, which causes a jump in membrane potential. Synaptic current and membrane potential decay exponentially with rates of alpha and beta, respectively. For $U[t] > U_{th} S[T + 1] = 1$.

If $reset\_mechanism = \text{"subtract"}$, then $U[t + 1]$ will have $threshold$ subtracted from it whenever the neuron emits a spike:

$$I_{syn}[t + 1] = I_{syn}[t] + V(S_{out}[t] + I_{in}[t + 1])$$
$$U[t + 1] = U[t] + I_{syn}[t + 1] - RU_{th}$$

Where $V(\cdot)$ acts either as a linear layer, a convolutional operator, or elementwise product on $S_{out}$.

- If $all\_to\_all = \text{"True"}$ and $linear\_features$ is specified, then $V(\cdot)$ acts as a recurrent linear layer of the same size as $S_{out}$.
• If \( \text{all}_\text{to}_\text{all} = \text{"True"} \) and \( \text{conv2d}_\text{channels} \) and \( \text{kernel}_\text{size} \) are specified, then \( V(\cdot) \) acts as a recurrent convolutional layer with padding to ensure the output matches the size of the input.

• If \( \text{all}_\text{to}_\text{all} = \text{"False"} \), then \( V(\cdot) \) acts as an elementwise multiplier with \( V \).

If \( \text{reset}_\text{mechanism} = \text{"zero"} \), then \( U[t + 1] \) will be set to \( 0 \) whenever the neuron emits a spike:

\[
I_{\text{syn}}[t + 1] = I_{\text{syn}}[t] + V S_{\text{out}}[t] + I_{\text{in}}[t + 1] \\
U[t + 1] = U[t] + I_{\text{syn}}[t + 1] - R(U[t] + I_{\text{syn}}[t + 1])
\]

- \( I_{\text{syn}} \) - Synaptic current
- \( I_{\text{in}} \) - Input current
- \( U \) - Membrane potential
- \( U_{\text{thr}} \) - Membrane threshold
- \( S_{\text{out}} \) - Output spike
- \( R \) - Reset mechanism: if active, \( R = 1 \), otherwise \( R = 0 \)

• - Synaptic current decay rate
• - Membrane potential decay rate
• - \( V \) - Explicit recurrent weight

Example:

```python
import torch
import torch.nn as nn
import snntorch as snn

alpha = 0.9
beta = 0.5

# shared recurrent connection for a given layer
V1 = 0.5

# independent connection p/neuron
V2 = torch.rand(num_outputs)

# Define Network
class Net(nn.Module):
    def __init__(self):
        super().__init__()

        # initialize layers
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.RSynaptic(alpha=alpha, beta=beta, linear_features=num_hidden)
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        self.lif2 = snn.RSynaptic(alpha=alpha, beta=beta, all_to_all=False, V=V2)

    def forward(self, x, syn1, mem1, spk1, syn2, mem2):
        cur1 = self.fc1(x)
        spk1, syn1, mem1 = self.lif1(cur1, spk1, syn1, mem1)
        cur2 = self.fc2(spk1)
```

(continues on next page)
Parameters

- **alpha** *(float or torch.tensor)* – synaptic current decay rate. Clipped between 0 and 1 during the forward-pass. May be a single-valued tensor (i.e., equal decay rate for all neurons in a layer), or multi-valued (one weight per neuron).

- **beta** *(float or torch.tensor)* – membrane potential decay rate. Clipped between 0 and 1 during the forward-pass. May be a single-valued tensor (i.e., equal decay rate for all neurons in a layer), or multi-valued (one weight per neuron).

- **V** *(float or torch.tensor)* – Recurrent weights to scale output spikes, only used when all_to_all=False. Defaults to 1.

- **all_to_all** *(bool, optional)* – Enables output spikes to be connected in dense or convolutional recurrent structures instead of 1-to-1 connections. Defaults to True.

- **linear_features** *(int, optional)* – Size of each output sample. Must be specified if all_to_all=True and the input data is 1D. Defaults to None

- **conv2d_channels** *(int, optional)* – Number of channels in each output sample. Must be specified if all_to_all=True and the input data is 3D. Defaults to None

- **kernel_size** *(int or tuple)* – Size of the convolving kernel. Must be specified if all_to_all=True and the input data is 3D. Defaults to None

- **threshold** *(float, optional)* – Threshold for mem to reach in order to generate a spike S=1. Defaults to 1

- **spike_grad** *(surrogate gradient function from snntorch.surrogate, optional)* – Surrogate gradient for the term dS/dU. Defaults to None (corresponds to Heaviside surrogate gradient. See snntorch.surrogate for more options)

- **init_hidden** *(bool, optional)* – Instantiates state variables as instance variables. Defaults to False

- **inhibition** *(bool, optional)* – If True, suppresses all spiking other than the neuron with the highest state. Defaults to False

- **learn_alpha** *(bool, optional)* – Option to enable learnable alpha. Defaults to False

- **learn_beta** *(bool, optional)* – Option to enable learnable beta. Defaults to False

- **learn_recurrent** *(bool, optional)* – Option to enable learnable recurrent weights. Defaults to True

- **learn_threshold** *(bool, optional)* – Option to enable learnable threshold. Defaults to False

- **reset_mechanism** *(str, optional)* – Defines the reset mechanism applied to mem each time the threshold is met. Reset-by-subtraction: “subtract”, reset-to-zero: “zero, none: “none”. Defaults to “subtract”

- **state_quant** *(quantization function from snntorch.quant, optional)* – If specified, hidden states mem and syn are quantized to a valid state for the forward pass. Defaults to False

- **output** *(bool, optional)* – If True as well as init_hidden=True, states are returned when neuron is called. Defaults to False
Inputs: input_, spk_0, syn_0, mem_0

- **input_** of shape (batch, input_size): tensor containing input features
- **spk_0** of shape (batch, input_size): tensor containing output spike features
- **syn_0** of shape (batch, input_size): tensor containing input features
- **mem_0** of shape (batch, input_size): tensor containing the initial membrane potential for each element in the batch.

Outputs: spk_1, syn_1, mem_1

- **spk_1** of shape (batch, input_size): tensor containing the output spikes.
- **syn_1** of shape (batch, input_size): tensor containing the next synaptic current for each element in the batch.
- **mem_1** of shape (batch, input_size): tensor containing the next membrane potential for each element in the batch.

Learnable Parameters:

- **RSynaptic.alpha** (torch.Tensor) - optional learnable weights must be manually passed in, of shape I or (input_size).
- **RSynaptic.beta** (torch.Tensor) - optional learnable weights must be manually passed in, of shape I or (input_size).
- **RSynaptic.recurrent.weight** (torch.Tensor) - optional learnable weights are automatically generated if all_to_all=True. RSynaptic.recurrent stores a nn.Linear or nn.Conv2d layer depending on input arguments provided.
- **RSynaptic.V** (torch.Tensor) - optional learnable weights must be manually passed in, of shape I or (input_size). It is only used where all_to_all=False for 1-to-1 recurrent connections.
- **RSynaptic.threshold** (torch.Tensor) - optional learnable thresholds must be manually passed in, of shape I or " (input_size).

```
classmethod detach_hidden()

Returns the hidden states, detached from the current graph. Intended for use in truncated backpropagation through time where hidden state variables are instance variables.
```

```
forward(input_, spk=False, syn=False, mem=False)

Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.
```

```
classmethod reset_hidden()

Used to clear hidden state variables to zero. Intended for use where hidden state variables are instance variables.
```

```
training: bool
```

```python
class snntorch._neurons.rsynaptic.RecurrentOneToOne(V)
Bases: Module
```
forward($x$)

Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

**training**: bool

**snn.SConv2dLSTM**

class snntorch._neurons.sconv2dlstm.SConv2dLSTM(in_channels, out_channels, kernel_size, bias=True, max_pool=0, avg_pool=0, threshold=1.0, spike_grad=None, init_hidden=False, inhibition=False, learn_threshold=False, reset_mechanism='none', state_quant=False, output=False)

Bases: SpikingNeuron

A spiking 2d convolutional long short-term memory cell. Hidden states are membrane potential and synaptic current $\text{mem}$, $\text{syn}$, which correspond to the hidden and cell states $h$, $c$ in the original LSTM formulation.

The input is expected to be of size $(N, C_{in}, H_{in}, W_{in})$ where $N$ is the batch size.

Unlike the LSTM module in PyTorch, only one time step is simulated each time the cell is called.

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}\text{mem}_{t-1} + b_{hi})$$

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}\text{mem}_{t-1} + b_{hf})$$

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}\text{mem}_{t-1} + b_{hg})$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}\text{mem}_{t-1} + b_{ho})$$

$$\text{syn}_t = f_t c_{t-1} + i_t g_t$$

$$\text{mem}_t = o_t \tanh(\text{syn}_t)$$

where $\sigma$ is the sigmoid function, $\ast$ is the 2D cross-correlation operator and $\cdot$ is the Hadamard product. The output state $\text{mem}_{t+1}$ is thresholded to determine whether an output spike is generated. To conform to standard LSTM state behavior, the default reset mechanism is set to $\text{reset} = \text{"none"}$, i.e., no reset is applied. If this is changed, the reset is only applied to $\text{mem}_t$.

Options to apply max-pooling or average-pooling to the state $\text{mem}_t$ are also enabled. Note that it is preferable to apply pooling to the state rather than the spike, as it does not make sense to apply pooling to activations of 1’s and 0’s which may lead to random tie-breaking.

Padding is automatically applied to ensure consistent sizes for hidden states from one time step to the next.

At the moment, stride $\neq 1$ is not supported.

Example:

```python
import torch
import torch.nn as nn
import snntorch as snn
```
beta = 0.5

# Define Network

class Net(nn.Module):
    def __init__(self):
        super().__init__()

        in_channels = 1
        out_channels = 8
        out_channels = 16
        kernel_size = 3
        max_pool = 2
        avg_pool = 2
        flattened_input = 49 * 16
        num_outputs = 10
        beta = 0.5

        spike_grad_lstm = surrogate.straight_through_estimator()
        spike_grad_fc = surrogate.fast_sigmoid(slope=5)

        # initialize layers
        self.sclstm1 = snn.SConv2dLSTM(in_channels, out_channels, kernel_size, max_˓→pool=max_pool, spike_grad=spike_grad_lstm)
        self.sclstm2 = snn.SConv2dLSTM(out_channels, out_channels, kernel_size, avg_˓→pool=avg_pool, spike_grad=spike_grad_lstm)
        self.fc2 = nn.Linear(flattened_input, num_outputs)
        self.lif2 = snn.Leaky(beta=beta, spike_grad=spike_grad_fc)

    def forward(self, x, mem1, spk1, mem2):
        # Initialize hidden states and outputs at t=0
        syn1, mem1 = self.lif1.init_sconv2dlstm()
        syn2, mem2 = self.lif2.init_sconv2dlstm()
        mem3 = self.lif3.init_leaky()

        # Record the final layer
        spk3_rec = []
        mem3_rec = []

        for step in range(num_steps):
            spk1, syn1, mem1 = self.lif1(x, syn1, mem1)
            spk2, syn2, mem2 = self.lif2(spk1, syn2, h2)
            cur = self.fc1(spk2.flatten(1))
            spk3, mem3 = self.lif3(cur, mem3)

            spk3_rec.append(spk3)
            mem3_rec.append(mem3)

        return torch.stack(spk3_rec), torch.stack(mem3_rec)

Parameters

- **in_channels**: (int) – number of input channels
• **kernel_size** *(int, tuple, or list)* – Size of the convolving kernel

• **bias** *(bool, optional)* – If True, adds a learnable bias to the output. Defaults to True

• **max_pool** *(int, tuple, or list, optional)* – Applies max-pooling to the hidden state \( \text{mem} \) prior to thresholding if specified. Defaults to 0

• **avg_pool** *(int, tuple, or list, optional)* – Applies average-pooling to the hidden state \( \text{mem} \) prior to thresholding if specified. Defaults to 0

• **threshold** *(float, optional)* – Threshold for \( \text{mem} \) to reach in order to generate a spike \( S=1 \). Defaults to 1

• **spike_grad** *(surrogate gradient function from snntorch.surrogate, optional)* – Surrogate gradient for the term \( dS/dU \). Defaults to a straight-through-estimator

• **learn_threshold** *(bool, optional)* – Option to enable learnable threshold. Defaults to False

• **init_hidden** *(bool, optional)* – Instantiates state variables as instance variables. Defaults to False

• **inhibition** *(bool, optional)* – If True, suppresses all spiking other than the neuron with the highest state. Defaults to False

• **reset_mechanism** *(str, optional)* – Defines the reset mechanism applied to \( \text{mem} \) each time the threshold is met. Reset-by-subtraction: “subtract”, reset-to-zero: “zero, none: “none”. Defaults to “none”

• **state_quant** *(quantization function from snntorch.quant, optional)* – If specified, hidden states \( \text{mem} \) and \( \text{syn} \) are quantized to a valid state for the forward pass. Defaults to False

• **output** *(bool, optional)* – If True as well as \( \text{init_hidden}=\text{True} \), states are returned when neuron is called. Defaults to False

**Inputs:** input, syn_0, mem_0

- **input** of shape \((\text{batch}, \text{in_channels}, H, W)\): tensor containing input features

- **syn_0** of shape \((\text{batch}, \text{out_channels}, H, W)\): tensor containing the initial synaptic current (or cell state) for each element in the batch.

- **mem_0** of shape \((\text{batch}, \text{out_channels}, H, W)\): tensor containing the initial membrane potential (or hidden state) for each element in the batch.

**Outputs:** spk, syn_1, mem_1

- **spk** of shape \((\text{batch}, \text{out_channels}, H/pool, W/pool)\): tensor containing the output spike (avg_pool and max_pool scale if greater than 0.)

- **syn_1** of shape \((\text{batch}, \text{out_channels}, H, W)\): tensor containing the next synaptic current (or cell state) for each element in the batch

- **mem_1** of shape \((\text{batch}, \text{out_channels}, H, W)\): tensor containing the next membrane potential (or hidden state) for each element in the batch

**Learnable Parameters:**

- **SConv2dLSTM.conv.weight** *(torch.Tensor)* - the learnable weights, of shape \(((\text{in_channels} + \text{out_channels}), 4*\text{out_channels}, \text{kernel_size})\).
classmethod detach_hidden()

Returns the hidden states, detached from the current graph. Intended for use in truncated backpropagation through time where hidden state variables are instance variables.

forward(input_, syn=False, mem=False)

Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

static init_sconv2dlstm()

Used to initialize h and c as an empty SpikeTensor. init_flag is used as an attribute in the forward pass to convert the hidden states to the same as the input.

classmethod reset_hidden()

Used to clear hidden state variables to zero. Intended for use where hidden state variables are instance variables.

training: bool

snn.SLSTM

class snntorch._neurons.slstm.SLSTM(input_size, hidden_size, bias=True, threshold=1.0, spike_grad=None, init_hidden=False, inhibition=False, learn_threshold=False, reset_mechanism='none', state_quant=False, output=False)

Bases: SpikingNeuron

A spiking long short-term memory cell. Hidden states are membrane potential and synaptic current mem, syn, which correspond to the hidden and cell states h, c in the original LSTM formulation.

The input is expected to be of size (N, X) where N is the batch size.

Unlike the LSTM module in PyTorch, only one time step is simulated each time the cell is called.

\[
\begin{align*}
    i_t &= \sigma(W_{ii} x_t + b_{ii} + W_{hi} \text{mem}_{t-1} + b_{hi}) \\
    f_t &= \sigma(W_{if} x_t + b_{if} + W_{hf} \text{mem}_{t-1} + b_{hf}) \\
    g_t &= \tanh(W_{ig} x_t + b_{ig} + W_{hg} \text{mem}_{t-1} + b_{hg}) \\
    o_t &= \sigma(W_{io} x_t + b_{io} + W_{ho} \text{mem}_{t-1} + b_{ho}) \\
    \text{syn}_t &= f_t \text{syn}_{t-1} + i_t g_t \\
    \text{mem}_t &= o_t \tanh(\text{syn}_t)
\end{align*}
\]

where \( \sigma \) is the sigmoid function and \( \odot \) is the Hadamard product. The output state \( \text{mem}_{t+1} \) is thresholded to determine whether an output spike is generated. To conform to standard LSTM state behavior, the default reset mechanism is set to \( \text{reset} = \text{'none'} \), i.e., no reset is applied. If this is changed, the reset is only applied to \( h_t \).

Example:

```python
import torch
import torch.nn as nn
import snntorch as snn

(continues on next page)```
beta = 0.5

# Define Network
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        num_inputs = 784
        num_hidden1 = 1000
        num_hidden2 = 10

        spike_grad_lstm = surrogate.straight_through_estimator()

        # initialize layers
        self.slstm1 = snn.SLSTM(num_inputs, num_hidden1, spike_grad=spike_grad_lstm)
        self.slstm2 = snn.SLSTM(num_hidden1, num_hidden2, spike_grad=spike_grad_lstm)

    def forward(self, x):
        syn1, mem1 = self.slstm1.init_slstm()
        syn2, mem2 = self.slstm2.init_slstm()

        spk2_rec = []
        mem2_rec = []

        for step in range(num_steps):
            spk1, syn1, mem1 = self.slstm1(x.flatten(1), syn1, mem1)
            spk2, syn2, mem2 = self.slstm2(spk1, syn2, mem2)

            spk2_rec.append(spk2)
            mem2_rec.append(mem2)

        return torch.stack(spk2_rec), torch.stack(mem2_rec)

Parameters

- **input_size** (int) – number of expected features in the input \( x \)
- **hidden_size** (int) – the number of features in the hidden state \( mem \)
- **bias** (bool, optional) – If True, adds a learnable bias to the output. Defaults to True
- **threshold** (float, optional) – Threshold for \( h \) to reach in order to generate a spike \( S=1 \). Defaults to 1
- **spike_grad** (surrogate gradient function from snntorch.surrogate, optional) – Surrogate gradient for the term \( dS/dU \). Defaults to a straight-through-estimator
- **learn_threshold** (bool, optional) – Option to enable learnable threshold. Defaults to False
- **init_hidden** *(bool, optional)* – Instantiates state variables as instance variables. Defaults to False

- **inhibition** *(bool, optional)* – If True, suppresses all spiking other than the neuron with the highest state. Defaults to False

- **reset_mechanism** *(str, optional)* – Defines the reset mechanism applied to *mem* each time the threshold is met. Reset-by-subtraction: “subtract”, reset-to-zero: “zero”, none: “none”. Defaults to “none”

- **state_quant** *(quantization function from snntorch.quant, optional)* – If specified, hidden states *mem* and *syn* are quantized to a valid state for the forward pass. Defaults to False

- **output** *(bool, optional)* – If True as well as *init_hidden=True*, states are returned when neuron is called. Defaults to False

**Inputs: input_, syn_0, mem_0**

- *input_* of shape *(batch, input_size)*: tensor containing input features

- *syn_0* of shape *(batch, hidden_size)*: tensor containing the initial synaptic current (or cell state) for each element in the batch.

- *mem_0* of shape *(batch, hidden_size)*: tensor containing the initial membrane potential (or hidden state) for each element in the batch.

**Outputs: spk, syn_1, mem_1**

- *spk* of shape *(batch, hidden_size)*: tensor containing the output spike

- *syn_1* of shape *(batch, hidden_size)*: tensor containing the next synaptic current (or cell state) for each element in the batch

- *mem_1* of shape *(batch, hidden_size)*: tensor containing the next membrane potential (or hidden state) for each element in the batch

**Learnable Parameters:**

- **SLSTM.lstm_cell.weight_ih** *(torch.Tensor)* - the learnable input-hidden weights, of shape *(4*hidden_size, input_size)*

- **SLSTM.lstm_cell.weight_ih** *(torch.Tensor)* – the learnable hidden-hidden weights, of shape *(4*hidden_size, hidden_size)*

- **SLSTM.lstm_cell.bias_ih** – the learnable input-hidden bias, of shape *(4*hidden_size)*

- **SLSTM.lstm_cell.bias_hh** – the learnable hidden-hidden bias, of shape *(4*hidden_size)*

**classmethod detach_hidden()**

Returns the hidden states, detached from the current graph. Intended for use in truncated backpropagation through time where hidden state variables are instance variables.

**forward**(input_, syn=False, mem=False)**

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.
static init_slstm()

Used to initialize mem and syn as an empty SpikeTensor. init_flag is used as an attribute in the forward pass to convert the hidden states to the same as the input.

classmethod reset_hidden()

Used to clear hidden state variables to zero. Intended for use where hidden state variables are instance variables.

training: bool

snn.Synaptic

class snntorch._neurons.synaptic.Synaptic(alpha, beta, threshold=1.0, spike_grad=None, init_hidden=False, inhibition=False, learn_alpha=False, learn_beta=False, learn_threshold=False, reset_mechanism='subtract', state_quant=False, output=False)

Bases: LIF

2nd order leaky integrate and fire neuron model accounting for synaptic conductance. The synaptic current jumps upon spike arrival, which causes a jump in membrane potential. Synaptic current and membrane potential decay exponentially with rates of alpha and beta, respectively. For $U[T] > U_{\text{thr}}S[T + 1] = 1$.

If reset_mechanism = “subtract”, then $U[t + 1]$ will have threshold subtracted from it whenever the neuron emits a spike:

\[
I_{\text{syn}}[t + 1] = I_{\text{syn}}[t] + I_{\text{in}}[t + 1] \\
U[t + 1] = U[t] + I_{\text{syn}}[t + 1] - RU_{\text{thr}}
\]

If reset_mechanism = “zero”, then $U[t + 1]$ will be set to 0 whenever the neuron emits a spike:

\[
I_{\text{syn}}[t + 1] = I_{\text{syn}}[t] + I_{\text{in}}[t + 1] \\
U[t + 1] = U[t] + I_{\text{syn}}[t + 1] - R(U[t] + I_{\text{syn}}[t + 1])
\]

- $I_{\text{syn}}$ - Synaptic current
- $I_{\text{in}}$ - Input current
- $U$ - Membrane potential
- $U_{\text{thr}}$ - Membrane threshold
- $R$ - Reset mechanism: if active, $R = 1$, otherwise $R = 0$
- - Synaptic current decay rate
- - Membrane potential decay rate

Example:

```python
import torch
import torch.nn as nn
import snntorch as snn

alpha = 0.9
beta = 0.5
```

(continues on next page)
# Define Network

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

        # initialize layers
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.Synaptic(alpha=alpha, beta=beta)
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        self.lif2 = snn.Synaptic(alpha=alpha, beta=beta)

    def forward(self, x, syn1, mem1, spk1, syn2, mem2):
        cur1 = self.fc1(x)
        spk1, syn1, mem1 = self.lif1(cur1, syn1, mem1)
        cur2 = self.fc2(spk1)
        spk2, syn2, mem2 = self.lif2(cur2, syn2, mem2)
        return syn1, mem1, spk1, syn2, mem2, spk2
```

Parameters

- **alpha** *(float or torch.tensor)* – synaptic current decay rate. Clipped between 0 and 1 during the forward-pass. May be a single-valued tensor (i.e., equal decay rate for all neurons in a layer), or multi-valued (one weight per neuron).

- **beta** *(float or torch.tensor)* – membrane potential decay rate. Clipped between 0 and 1 during the forward-pass. May be a single-valued tensor (i.e., equal decay rate for all neurons in a layer), or multi-valued (one weight per neuron).

- **threshold** *(float, optional)* – Threshold for mem to reach in order to generate a spike $S=1$. Defaults to 1

- **spike_grad** *(surrogate gradient function from snntorch.surrogate, optional)* – Surrogate gradient for the term dS/dU. Defaults to None (corresponds to Heaviside surrogate gradient. See snntorch.surrogate for more options)

- **init_hidden** *(bool, optional)* – Instantiates state variables as instance variables. Defaults to False

- **inhibition** *(bool, optional)* – If True, suppresses all spiking other than the neuron with the highest state. Defaults to False

- **learn_alpha** *(bool, optional)* – Option to enable learnable alpha. Defaults to False

- **learn_beta** *(bool, optional)* – Option to enable learnable beta. Defaults to False

- **learn_threshold** *(bool, optional)* – Option to enable learnable threshold. Defaults to False

- **reset_mechanism** *(str, optional)* – Defines the reset mechanism applied to mem each time the threshold is met. Reset-by-subtraction: “subtract”, reset-to-zero: “zero, none: “none”. Defaults to “subtract”

- **state_quant** *(quantization function from snntorch.quant, optional)* – If specified, hidden states mem and syn are quantized to a valid state for the forward pass. Defaults to False

- **output** *(bool, optional)* – If True as well as init_hidden=True, states are returned when neuron is called. Defaults to False
Inputs: input_, syn_0, mem_0

- input_ of shape (batch, input_size): tensor containing input features
- syn_0 of shape (batch, input_size): tensor containing input features
- mem_0 of shape (batch, input_size): tensor containing the initial membrane potential for each element in the batch.

Outputs: spk, syn_1, mem_1

- spk of shape (batch, input_size): tensor containing the output spikes.
- syn_1 of shape (batch, input_size): tensor containing the next synaptic current for each element in the batch.
- mem_1 of shape (batch, input_size): tensor containing the next membrane potential for each element in the batch.

Learnable Parameters:

- Synaptic.alpha (torch.Tensor): optional learnable weights must be manually passed in, of shape 1 or (input_size).
- Synaptic.beta (torch.Tensor): optional learnable weights must be manually passed in, of shape 1 or (input_size).
- Synaptic.threshold (torch.Tensor): optional learnable thresholds must be manually passed in, of shape 1 or (input_size).

classmethod detach_hidden()

Returns the hidden states, detached from the current graph. Intended for use in truncated backpropagation through time where hidden state variables are instance variables.

forward(input_, syn=False, mem=False)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

classmethod reset_hidden()

Used to clear hidden state variables to zero. Intended for use where hidden state variables are instance variables.

training: bool
Neuron Parent Classes

class snntorch._neurons.neurons.LIF(
    beta, threshold=1.0, spike_grad=None, init_hidden=False,
    inhibition=False, learn_beta=False, learn_threshold=False,
    reset_mechanism='subtract', state_quant=False, output=False)

Bases: SpikingNeuron

Parent class for leaky integrate and fire neuron models.

static init_alpha()
    Used to initialize syn_exc, syn_inh and mem as an empty SpikeTensor. init_flag is used as an attribute in the forward pass to convert the hidden states to the same as the input.

static init_lapicque()
    Used to initialize mem as an empty SpikeTensor. init_flag is used as an attribute in the forward pass to convert the hidden states to the same as the input.

static init_leaky()
    Used to initialize mem as an empty SpikeTensor. init_flag is used as an attribute in the forward pass to convert the hidden states to the same as the input.

static init_rleaky()
    Used to initialize spk and mem as an empty SpikeTensor. init_flag is used as an attribute in the forward pass to convert the hidden states to the same as the input.

static init_rsynaptic()
    Used to initialize spk, syn and mem as an empty SpikeTensor. init_flag is used as an attribute in the forward pass to convert the hidden states to the same as the input.

static init_synaptic()
    Used to initialize syn and mem as an empty SpikeTensor. init_flag is used as an attribute in the forward pass to convert the hidden states to the same as the input.

training: bool

class snntorch._neurons.neurons.SpikingNeuron(
    threshold=1.0, spike_grad=None, init_hidden=False,
    inhibition=False, learn_threshold=False,
    reset_mechanism='subtract', state_quant=False, output=False)

Bases: Module

Parent class for spiking neuron models.

class ATan(*args, **kwargs)
    Bases: Function

Surrogate gradient of the Heaviside step function.

Forward pass: Heaviside step function shifted.

\[ S = \begin{cases} 1 & \text{if } U > U_{th} \\ 0 & \text{if } U < U_{th} \end{cases} \]

Backward pass: Gradient of shifted arc-tan function.
\[ S^{-1} \arctan(U_2) \]
\[ S = \frac{1}{1 + (U_2)^2} \]

`alpha` defaults to 2, and can be modified by calling `surrogate.atan(alpha=2)`.

Adapted from:


**static backward(ctx, grad_output)**

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by as many outputs as the `forward()` returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to `forward()`. Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx.needs_input_grad` as a tuple of booleans representing whether each input needs gradient. E.g., `backward()` will have `ctx.needs_input_grad[0] = True` if the first input to `forward()` needs gradient computed w.r.t. the output.

**static forward(ctx, input_, alpha=2.0)**

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on `ctx` (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with `ctx.save_for_backward()` if they are intended to be used in `backward` (equivalently, vjp) or `ctx.save_for_forward()` if they are intended to be used for in jvp.

**static detach(*args)**

Used to detach input arguments from the current graph. Intended for use in truncated backpropagation through time where hidden state variables are global variables.

**fire(mem)**

Generates spike if mem > threshold. Returns spk.

**fire_inhibition(batch_size, mem)**

Generates spike if mem > threshold, only for the largest membrane. All others neurons will be inhibited for that time step. Returns spk.

**classmethod init()**

Removes all items from `snntorch.SpikingNeuron.instances` when called.
instances = []

Each `snntorch.SpikingNeuron` neuron (e.g., `snntorch.Synaptic`) will populate the `snntorch.SpikingNeuron.instances` list with a new entry. The list is used to initialize and clear neuron states when the argument `init_hidden=True`.

```python
mem_reset(mem)
```
Generates detached reset signal if mem > threshold. Returns reset.

```python
reset_dict = {'none': 2, 'subtract': 0, 'zero': 1}
```

```python
property reset_mechanism
```
If reset_mechanism is modified, reset_mechanism_val is triggered to update. 0: subtract, 1: zero, 2: none.

```python
training: bool
```

```python
static zeros(*args)
```
Used to clear hidden state variables to zero. Intended for use where hidden state variables are global variables.

## 1.11.4 `snntorch.backprop`

`snntorch.backprop` is a module implementing various time-variant backpropagation algorithms. Each method will perform the forward-pass, backward-pass, and parameter update across all time steps in a single line of code.

### How to use backprop

To use `snntorch.backprop` you must first construct a network, determine a loss criterion, and select an optimizer. When initializing neurons, set `init_hidden=True`. This enables the methods in `snntorch.backprop` to automatically clear the hidden state variables, as well as detach them from the computational graph when necessary.

**Note:** The first dimension of input data is assumed to be time. The built-in backprop functions iterate through the first dimension of data by default. For time-invariant inputs, set `time_var=False`.

Example:

```python
net = Net().to(device)
optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas=betas)
criterion = nn.CrossEntropyLoss()

# Time-variant input data
for input, target in dataset:
    loss = BPTT(net, input, target, num_steps, batch_size, optimizer, criterion)

# Time-invariant input data
for input, targets in dataset:
    loss = BPTT(net, input, target, num_steps, batch_size, optimizer, criterion, time_var=False)
```

### `snntorch.backprop.BPTT`

```python
snntorch.backprop.BPTT(net, dataloader, optimizer, criterion, num_steps=False, time_var=True,
    time_first=True, regularization=False, device='cpu')
```

Backpropagation through time. LIF layers require parameter `init_hidden = True`. A forward pass is applied for each time step while the loss accumulates. The backward pass and parameter update is only applied at the end of each time step sequence. BPTT is equivalent to TBPTT for the case where `num_steps = K`. 

---

Chapter 1. Introduction
Example:

```python
import snntorch as snn
import snntorch.functional as SF
from snntorch import utils
from snntorch import backprop
import torch
import torch.nn as nn

lif1 = snn.Leaky(beta=0.9, init_hidden=True)
lif2 = snn.Leaky(beta=0.9, init_hidden=True, output=True)

net = nn.Sequential(nn.Flatten(),
                     nn.Linear(784, 500),
                     lif1,
                     nn.Linear(500, 10),
                     lif2).to(device)

device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
num_steps = 100

optimizer = torch.optim.Adam(net.parameters(), lr=5e-4, betas=(0.9, 0.999))
loss_fn = SF.mse_count_loss()
reg_fn = SF.l1_rate_sparsity()

# train_loader is of type torch.utils.data.DataLoader
# if input data is time-static, set time_var=False, and specify num_steps.
# if input data is time-varying, set time_var=True and do not specify num_steps.

for epoch in range(5):
    loss = backprop.RTRL(net, train_loader, optimizer=optimizer,
                         criterion=loss_fn, num_steps=num_steps, time_var=False,
                         regularization=reg_fn, device=device)
```

Parameters

- **net** (`torch.nn.modules.container.Sequential`) – Network model (either wrapped in `Sequential` container or as a class)
- **dataloader** (`torch.utils.data.DataLoader`) – DataLoader containing data and targets
- **optimizer** (`torch.optim`) – Optimizer used, e.g., `torch.optim.adam.Adam`
- **criterion** (`snn.functional.LossFunctions`) – Loss criterion from `snntorch.functional`, e.g., `snn.functional.mse_count_loss()`
- **num_steps** (`int`, optional) – Number of time steps. Does not need to be specified if `time_var=True`.
- **time_var** (`Bool`, optional) – Set to True if input data is time-varying [T x B x dims]. Otherwise, set to false if input data is time-static [B x dims], defaults to True
- **time_first** (`Bool`, optional) – Set to False if first dimension of data is not time [B x T x dims] AND must also be permuted to [T x B x dims], defaults to False
- **regularization** (`snn.functional regularization function`, optional) – Option to add a regularization term to the loss function
**device** *(string, optional)* – Specify either “cuda” or “cpu”, defaults to “cpu”

**Returns**

return average loss for one epoch

**Return type**

torch.Tensor

```python
snntorch.backprop.RTRL(net, dataloader, optimizer, criterion, num_steps=False, time_var=True,
time_first=True, regularization=False, device='cpu')
```

Real-time Recurrent Learning. LIF layers require parameter `init_hidden = True`. A forward pass, backward pass and parameter update are applied at each time step. RTRL is equivalent to TBPTT for the case where $K = 1$.

Example:

```python
import snntorch as snn
import snntorch.functional as SF
from snntorch import utils
from snntorch import backprop
import torch
import torch.nn as nn

lif1 = snn.Leaky(beta=0.9, init_hidden=True)
lif2 = snn.Leaky(beta=0.9, init_hidden=True, output=True)

net = nn.Sequential(nn.Flatten(),
                     nn.Linear(784, 500),
                     lif1,
                     nn.Linear(500, 10),
                     lif2).to(device)

device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
num_steps = 100

optimizer = torch.optim.Adam(net.parameters(), lr=5e-4, betas=(0.9, 0.999))
loss_fn = SF.mse_count_loss()
reg_fn = SF.l1_rate_sparsity()

# train_loader is of type torch.utils.data.DataLoader
# if input data is time-static, set time_var=False, and specify num_steps.

for epoch in range(5):
    loss = backprop.RTRL(net, train_loader, optimizer=optimizer,
                          criterion=loss_fn, num_steps=num_steps, time_var=False,
                          regularization=reg_fn, device=device)
```

**Parameters**

- **net** *(torch.nn.modules.container.Sequential)* – Network model (either wrapped in Sequential container or as a class)

- **dataloader** *(torch.utils.data.DataLoader)* – DataLoader containing data and targets

- **optimizer** *(torch.optim)* – Optimizer used, e.g., torch.optim.adam.Adam
• **criterion** (*snn.functional.LossFunctions*) – Loss criterion from snntorch.functional, e.g., snn.functional.mse_count_loss()

• **num_steps** (*int, optional*) – Number of time steps. Does not need to be specified if `time_var=True`.

• **time_var** (*Bool, optional*) – Set to True if input data is time-varying [T x B x dims]. Otherwise, set to false if input data is time-static [B x dims], defaults to True.

• **time_first** (*Bool, optional*) – Set to False if first dimension of data is not time [B x T x dims] AND must also be permuted to [T x B x dims], defaults to True.

• **regularization** (*snn.functional regularization function, optional*) – Option to add a regularization term to the loss function.

• **device** (*string, optional*) – Specify either “cuda” or “cpu”, defaults to “cpu”.

• **K** (*int, optional*) – Number of time steps to process per weight update, defaults to 1.

**Returns**

return average loss for one epoch

**Return type**

torch.Tensor

```python
snntorch.backprop.TBPTT(net, dataloader, optimizer, criterion, num_steps=False, time_var=True,
                         time_first=True, regularization=False, device='cpu', K=1)
```

Truncated backpropagation through time. LIF layers require parameter `init_hidden = True`. Weight updates are performed every K time steps.

**Example:**

```python
import snntorch as snn
import snntorch.functional as SF
from snntorch import utils
from snntorch import backprop
import torch
import torch.nn as nn

lif1 = snn.Leaky(beta=0.9, init_hidden=True)
lif2 = snn.Leaky(beta=0.9, init_hidden=True, output=True)

net = nn.Sequential(nn.Flatten(),
                     nn.Linear(784,500),
                     lif1,
                     nn.Linear(500, 10),
                     lif2).to(device)

device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
num_steps = 100

optimizer = torch.optim.Adam(net.parameters(), lr=5e-4, betas=(0.9, 0.999))
loss_fn = SF.mse_count_loss()
reg_fn = SF.l1_rate_sparsity()

# train_loader is of type torch.utils.data.DataLoader
# if input data is time-static, set time_var=False, and specify num_steps.
```

(continues on next page)
# if input data is time-varying, set time_var=True and do not specify num_steps.
# backprop is automatically applied every K=40 time steps

```python
for epoch in range(5):
    loss = backprop.RTRL(net, train_loader, optimizer=optimizer,
                          criterion=loss_fn, num_steps=num_steps, time_var=False,
                          regularization=reg_fn, device=device, K=40)
```

Parameters

- **net** (torch.nn.modules.container.Sequential) – Network model (either wrapped in Sequential container or as a class)
- **dataloader** (torch.utils.data.DataLoader) – DataLoader containing data and targets
- **optimizer** (torch.optim) – Optimizer used, e.g., torch.optim.adam.Adam
- **criterion** (snn.functional.LossFunctions) – Loss criterion from snn.functional, e.g., snn.functional.mse_count_loss()
- **num_steps** (int, optional) – Number of time steps. Does not need to be specified if time_var=True.
- **time_var** (Bool, optional) – Set to True if input data is time-varying [T x B x dims]. Otherwise, set to false if input data is time-static [B x dims], defaults to True
- **time_first** (Bool, optional) – Set to False if first dimension of data is not time [B x T x dims] AND must also be permuted to [T x B x dims], defaults to True
- **regularization** (snn.functional regularization function, optional) – Option to add a regularization term to the loss function
- **device** (string, optional) – Specify either “cuda” or “cpu”, defaults to “cpu”
- **K** (int, optional) – Number of time steps to process per weight update, defaults to 1

Returns

return average loss for one epoch

Return type

torch:Tensor

1.11.5 snntorch.functional

snntorch.functional implements common arithmetic operations applied to spiking neurons, such as loss and regularization functions, and state quantization etc.
How to use functional

To use snntorch.functional you assign the function state to a variable, and then call that variable.

Example:

```python
import snntorch as snn
import snntorch.functional as SF

net = Net().to(device)
optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas=betas)
criterion = SF.ce_count_loss()  # apply cross-entropy to spike count

spk_rec, mem_rec = net(input_data)
loss = loss_fn(spk_rec, targets)

optimizer.zero_grad()
loss.backward()

# Weight Update
optimizer.step()
```

Accuracy Functions

snntorch.functional.acc.accuracy_rate(spk_out, targets, population_code=False, num_classes=False)

Use spike count to measure accuracy.

Parameters

- **spk_out** (*torch.Tensor*) – Output spikes of shape [num_steps x batch_size x num_outputs]
- **targets** (*torch.Tensor*) – Target tensor (without one-hot-encoding) of shape [batch_size]

Returns

- **accuracy**

Return type

- **numpy.float64**

snntorch.functional.acc.accuracy_temporal(spk_out, targets)

Loss Functions

class snntorch.functional.loss.LossFunctions

Bases: object

class snntorch.functional.loss.SpikeTime(target_is_time=False, on_target=0, off_target=-1, tolerance=0, multi_spike=False)

Bases: Module

Used by ce_temporal_loss and mse_temporal_loss to convert spike outputs into spike times.

class FirstSpike(*args, **kwargs)

Bases: Function

Convert spk_rec of 1/0s [TxBxN] -> first spike time [BxN]. Linearize df/dS=-1 if spike, 0 if no spike.
static backward(ctx, grad_output)

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by as many outputs as the forward() returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to forward(). Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute ctx. needs_input_grad as a tuple of booleans representing whether each input needs gradient. E.g., backward() will have ctx.needs_input_grad[0] = True if the first input to forward() needs gradient computed w.r.t. the output.

static forward(ctx, spk_rec, device='cpu')

Convert spk_rec of 1/0s [TxBxN] -> spk_time [TxNxT]. 0’s indicate no spike -> +1 is first time step. Transpose accounts for broadcasting along final dimension (i.e., multiply along T).

class MultiSpike(*args, **kwargs)

Bases: Function

Convert spk_rec of 1/0s [TxBxN] --> first F spike times [FxNxT]. Linearize df/dS=-1 if spike, 0 if no spike.

static backward(ctx, grad_output)

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by as many outputs as the forward() returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to forward(). Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute ctx. needs_input_grad as a tuple of booleans representing whether each input needs gradient. E.g., backward() will have ctx.needs_input_grad[0] = True if the first input to forward() needs gradient computed w.r.t. the output.

static forward(ctx, spk_rec, spk_count, device='cpu')

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on ctx (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with ctx.save_for_backward() if they are intended to be used in backward (equivalently, vjp) or ctx.save_for_forward() if they are intended to be used for in jvp.
class Tolerance(*args, **kwargs)

Bases: Function

If spike time is ‘close enough’ to target spike within tolerance, set the time to target for loss calc only.

static backward(ctx, grad_output)

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by as many outputs as the forward() returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to forward(). Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute ctx. needs_input_grad as a tuple of booleans representing whether each input needs gradient. E.g., backward() will have ctx.needs_input_grad[0] = True if the first input to forward() needs gradient computed w.r.t. the output.

static forward(ctx, spk_time, target, tolerance)

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on ctx (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with ctx.save_for_backward() if they are intended to be used in backward (equivalently, vjp) or ctx.save_for_forward() if they are intended to be used for in jvp.

forward(spk_out, targets)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

label_to_multi_spike(targets, num_outputs)

Convert labels from neuron index (dim: B) to multiple spike times (dim: F x B x N). F is the number of spikes per neuron. Assumes target is iterable along F.

label_to_single_spike(targets, num_outputs)

Convert labels from neuron index (dim: B) to first spike time (dim: B x N).

labels_to_spike_times(targets, num_outputs)

Convert index labels [B] into spike times.

training: bool
class snntorch.functional.loss.ce_count_loss(population_code=False, num_classes=False)

Bases: LossFunctions

Cross Entropy Spike Count Loss.

The spikes at each time step [num_steps x batch_size x num_outputs] are accumulated and then passed through
the Cross Entropy Loss function. This criterion combines log_softmax and NLLLoss in a single function. The
Cross Entropy Loss encourages the correct class to fire at all time steps, and aims to suppress incorrect classes
from firing.

The Cross Entropy Count Loss accumulates spikes first, and applies Cross Entropy Loss only once. In contrast,
the Cross Entropy Rate Loss applies the Cross Entropy function at every time step.

Example:

```python
import snntorch.functional as SF

# if not using population codes (i.e., more output neurons than there are classes)
loss_fn = ce_count_loss()
loss = loss_fn(spk_out, targets)

# if using population codes; e.g., 200 output neurons, 10 output classes --> 20 neurons p/class
loss_fn = ce_count_loss(population_code=True, num_classes=10)
loss = loss_fn(spk_out, targets)
```

Parameters

- **population_code** (bool, optional) – Specify if a population code is applied, i.e., the
  number of outputs is greater than the number of classes. Defaults to False

- **num_classes** (int, optional) – Number of output classes must be specified if
  population_code=True. Must be a factor of the number of output neurons if population
code is enabled. Defaults to False

Returns

Loss

Return type

torch.Tensor (single element)

class snntorch.functional.loss.ce_max_membrane_loss

Bases: LossFunctions

Cross Entropy Max Membrane Loss. When called, the maximum membrane potential value for each output
neuron is sampled and passed through the Cross Entropy Loss Function. This criterion combines log_softmax
and NLLLoss in a single function. The Cross Entropy Loss encourages the maximum membrane potential of the
correct class to increase, while suppressing the maximum membrane potential of incorrect classes. This function
is adopted from SpyTorch by Friedemann Zenke.

Example:

```python
import snntorch.functional as SF

loss_fn = SF.ce_max_membrane_loss()
loss = loss_fn(outputs, targets)
```
**Returns**
Loss

**Return type**
torch.Tensor (single element)

class snntorch.functional.loss.ce_rate_loss

Bases: LossFunctions

Cross Entropy Spike Rate Loss. When called, the spikes at each time step are sequentially passed through the CrossEntropyLoss function. This criterion combines log_softmax and NLLLoss in a single function. The losses are accumulated over time steps to give the final loss. The Cross Entropy Loss encourages the correct class to fire at all time steps, and aims to suppress incorrect classes from firing.

The Cross Entropy Rate Loss applies the Cross Entropy function at every time step. In contrast, the Cross Entropy Count Loss accumulates spikes first, and applies Cross Entropy Loss only once.

Example:

```python
import snntorch.functional as SF
loss_fn = SF.ce_rate_loss()
loss = loss_fn(outputs, targets)
```

**Returns**
Loss

**Return type**
torch.Tensor (single element)

class snntorch.functional.loss.ce_temporal_loss

Bases: object

Cross Entropy Temporal Loss.

The cross entropy loss of an ‘inverted’ first spike time of each output neuron [batch_size x num_outputs] is calculated. The ‘inversion’ is applied such that maximizing the value of the correct class decreases the first spike time (i.e., earlier spike).

Options for inversion include: inverse='negate' which applies (-1 * output), or inverse='reciprocal' which takes (1/output).

Note that the derivative of each spike time with respect to the spike df/dU is non-differentiable for most neuron classes, and is set to a sign estimator of -1. I.e., increasing membrane potential causes a proportionately earlier firing time.

Index labels are passed as the target. To specify the exact spike time, use mse_temporal_loss instead.

Note: After spike times with specified targets, no penalty is applied for subsequent spiking.

Example:

```python
import torch
import snntorch.functional as SF

# correct classes aimed to fire by default at t=0, incorrect at final step
loss_fn = ce_temporal_loss()
loss = loss_fn(spk_out, targets)
```
Parameters

inverse (str, optional) – Specify how to invert output before taking cross entropy. Either scale by \((-1 \times x)\) with inverse='negate' or take the reciprocal \((1/x)\) with inverse='reciprocal'. Defaults to negate

Returns

Loss

Return type

torch.Tensor (single element)

class snntorch.functional.loss.mse_count_loss(correct_rate=1, incorrect_rate=0, population_code=False, num_classes=False)

Bases: LossFunctions

Mean Square Error Spike Count Loss. When called, the total spike count is accumulated over time for each neuron. The target spike count for correct classes is set to \((\text{num\_steps} \times \text{correct\_rate})\), and for incorrect classes \((\text{num\_steps} \times \text{incorrect\_rate})\). The spike counts and target spike counts are then applied to a Mean Square Error Loss Function. This function is adopted from SLAYER by Sumit Bam Shrestha and Garrick Orchard.

Example:

```python
import snntorch.functional as SF

loss_fn = SF.mse_count_loss(correct_rate=0.75, incorrect_rate=0.25)
loss = loss_fn(outputs, targets)
```

Parameters

- **correct_rate** (float, optional) – Firing frequency of correct class as a ratio, e.g., 1 promotes firing at every step; 0.5 promotes firing at 50% of steps, 0 discourages any firing, defaults to 1
- **incorrect_rate** (float, optional) – Firing frequency of incorrect class(es) as a ratio, e.g., 1 promotes firing at every step; 0.5 promotes firing at 50% of steps, 0 discourages any firing, defaults to 1
- **population_code** (bool, optional) – Specify if a population code is applied, i.e., the number of outputs is greater than the number of classes. Defaults to False
- **num_classes** (int, optional) – Number of output classes must be specified if population_code=True. Must be a factor of the number of output neurons if population code is enabled. Defaults to False

Returns

Loss

Return type

torch.Tensor (single element)

class snntorch.functional.loss.mse_membrane_loss(time_var_targets=False, on_target=1, off_target=0)

Bases: LossFunctions

Mean Square Error Membrane Loss. When called, pass the output membrane of shape \([\text{num\_steps} \times \text{batch\_size} \times \text{num\_outputs}]\) and the target tensor of membrane potential. The membrane potential and target are then applied to a Mean Square Error Loss Function. This function is adopted from Spike-Op by Jason K. Eshraghian.

Example:
import snntorch.functional as SF

# if targets are the same at each time-step
loss_fn = mse_membrane_loss(time_var_targets=False)
loss = loss_fn(outputs, targets)

# if targets are time-varying
loss_fn = mse_membrane_loss(time_var_targets=True)
loss = loss_fn(outputs, targets)

Parameters

- **time_var_targets** – Specifies whether the targets are time-varying, defaults to False

- **on_target** *(float, optional)* – Specify target membrane potential for correct class, defaults to 1

- **off_target** *(float, optional)* – Specify target membrane potential for incorrect class, defaults to 0

Returns

Loss

Return type

torch.Tensor (single element)

class snntorch.functional.loss.mse_temporal_loss(target_is_time=False, on_target=0, off_target=-1, tolerance=0, multi_spike=False)

Bases: object

Mean Square Error Temporal Loss.

The first spike time of each output neuron [batch_size x num_outputs] is measured against the desired spike time with the Mean Square Error Loss Function. Note that the derivative of each spike time with respect to the spike df/dU is non-differentiable for most neuron classes, and is set to a sign estimator of -1. I.e., increasing membrane potential causes a proportionately earlier firing time.

The Mean Square Error Temporal Loss can account for multiple spikes by setting *multi_spike=True*. If the actual spike time is close enough to the target spike time within a given tolerance, e.g., *tolerance = 5* time steps, then it does not contribute to the loss.

Index labels are passed as the target by default. To enable passing in the spike time(s) for output neuron(s), set *target_is_time=True*.

Note: After spike times with specified targets, no penalty is applied for subsequent spiking. To eliminate later spikes, an additional target should be applied.

Example:

```python
import torch
import snntorch.functional as SF

# default takes in idx labels as targets
# correct classes aimed to fire by default at t=0, incorrect at t=-1 (final time_step)
loss_fn = mse_temporal_loss()
loss = loss_fn(spk_out, targets)
```
# as above, but correct class fire @ t=5, incorrect at t=100 with a tolerance of 2 steps
loss_fn = mse_temporal_loss(on_target=5, off_target=100, tolerance=2)
loss = loss_fn(spk_out, targets)

# as above with multiple spike time targets
on_target = torch.tensor(5, 10)
off_target = torch.tensor(100, 105)
loss_fn = mse_temporal_loss(on_target=on_target, off_target=off_target, tolerance=2)
loss = loss_fn(spk_out, targets)

# specify first spike time for 5 neurons individually, zero tolerance
target = torch.tensor(5, 10, 15, 20, 25)
loss_fn = mse_temporal_loss(target_is_time=True)
loss = loss_fn(spk_out, target)

Parameters

- **target_is_time** (bool, optional) – Specify if target is specified as spike times (True) or as neuron indexes (False). Defaults to False
- **on_target** (int or iterable over multiple int if multi_spike=True, optional) – Spike time for correct classes (only if target_is_time=False). Defaults to 0
- **off_target** (int or iterable over multiple int if multi_spike=True, optional) – Spike time for incorrect classes (only if target_is_time=False). Defaults to -1, i.e., final time step
- **tolerance** (int, optional) – If the distance between the spike time and target is less than the specified tolerance, then it does not contribute to the loss. Defaults to 0.
- **multi_spike** (bool, optional) – Specify if multiple spikes in target. Defaults to False

Returns

Loss

Return type
torch.Tensor (single element)

Regularization Functions

class snntorch.functional.reg.l1_rate_sparsity(Lambda=1e-05)
  Bases: object
  L1 regularization using total spike count as the penalty term. Lambda is a scalar factor for regularization.
State Quantization

class snntorch.functional.quant.StateQuant(*args, **kwargs)

Bases: Function

Wrapper function for state_quant

static backward(ctx, grad_output)

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by as many outputs as the forward() returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to forward(). Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute ctx.needs_input_grad as a tuple of booleans representing whether each input needs gradient. E.g., backward() will have ctx.needs_input_grad[0] = True if the first input to forward() needs gradient computed w.r.t. the output.

static forward(ctx, input_, levels)

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on ctx (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with ctx.save_for_backward() if they are intended to be used in backward (equivalently, vjp) or ctx.save_for_forward() if they are intended to be used for in jvp.

snntorch.functional.quant.state_quant(num_bits=8, uniform=True, thr_centered=True, threshold=1, lower_limit=0, upper_limit=0.2, multiplier=None)

Quantization-Aware Training with spiking neuron states.

Note: for weight quantization, we recommend using Brevitas or another pre-existing PyTorch-friendly library.

Uniform and non-uniform quantization can be applied in various modes by specifying uniform=True.

Valid quantization levels can be centered about 0 or threshold by specifying thr_centered=True.

upper_limit and lower_limit specify the proportion of how far valid levels go above and below the positive and negative threshold. E.g., upper_limit=0.2 means the maximum valid state is 20% higher than the value specified in threshold.

Example:

```python
import torch
import snntorch as snn
from snntorch.functional import quant

beta = 0.5
thr = 5
```
# set the quantization parameters
q_lif = quant.state_quant(num_bits=4, uniform=True, threshold=thr)

# specifying state_quant applies state-quantization to the hidden state(s)
lif = snn.Leaky(beta=beta, threshold=thr, state_quant=q_lif)

rand_input = torch.rand(1)
mem = lif.init_leaky()

# forward-pass for one step
spk, mem = lif(rand_input, mem)

Note: Quantization-Aware training is focused on modelling a reduced precision network, but does not in of itself accelerate low-precision models. Hidden states are still represented as full precision values for compatibility with PyTorch. For accelerated performance or constrained-memory, the model should be exported to a downstream backend.

Parameters

- **num_bits** *(int, optional)* – Number of bits to quantize state variables to, defaults to 8
- **uniform** *(Bool, optional)* – Applies uniform quantization if specified, non-uniform if unspecified, defaults to True
- **uniform** – For non-uniform quantization, specifies if valid states should be centered (densely clustered) around the threshold rather than at 0, defaults to True
- **threshold** *(float, optional)* – Specifies the threshold, defaults to 1
- **lower_limit** *(float, optional)* – Specifies how far below (-threshold) the lowest valid state can be, i.e., (-threshold - threshold*lower_limit), defaults to 0
- **upper_limit** *(float, optional)* – Specifies how far above (threshold) the highest valid state can be, i.e., (threshold + threshold*upper_limit), defaults to 0.2
- **multiplier** *(float, optional)* – For non-uniform distributions, specify the base of the exponential. If None, an appropriate value is set internally based on num_bits, defaults to None

Probe

```python

Bases: BaseMonitor
```

A monitor to record the attribute (e.g., membrane potential) of a specific neuron layer (e.g., Leaky) in a network. The attribute name can be specified as the first argument of this function. All attribute data is recorded in self. record as data type “list”. Call self.enable() or self.disable() to enable or disable the monitor. Call self.clear_recorded_data() to clear recorded data.

Example:
```python
import snntorch as snn
from snntorch.functional import probe

import torch
from torch import nn

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(8, 4)
        self.lif1 = snn.Leaky()
        self.fc2 = nn.Linear(4, 2)
        self.lif2 = snn.Leaky()

    def forward(self, x_seq: torch.Tensor):
        x_seq = self.fc1(x_seq)
        x_seq = self.lif1(x_seq)
        x_seq = self.fc2(x_seq)
        x_seq = self.lif2(x_seq)
        return x_seq

net = Net()
monitor = probe.AttributeMonitor('mem', False, net, instance=snn.Leaky())

with torch.no_grad():
    y = net(torch.rand([1, 8]))
    print(f'monitor.records={monitor.records}"
    print(f'monitor[0]="{monitor[0]}"
    print(f'monitor.monitored_layers={monitor.monitored_layers}"
    print(f"monitor['lif1']={monitor['lif1']}"

Parameters

- **attribute_name** – Attribute’s name of probed neuron layer (e.g., mem, syn, etc.)
- **pre_forward** *(bool)* – If True, record the attribute value before the forward pass, otherwise record the value after forward pass.
- **net** *(nn.Module)* – Network model (either wrapped in Sequential container or as a class)
- **instance** *(Any or tuple)* – Instance of modules to be monitored. If None, defaults to type(net)
- **function_on_attribute** *(Callable, optional)* – Function that is applied to the monitored modules’ attribute

create_hook(name)
```

1.11. License & Copyright 55
enable()

is_enable()

remove_hooks()


Bases: BaseMonitor

A monitor to record the input gradient of each neuron layer (e.g. Leaky) in a network. All input gradient data is recorded in self.record as data type “list”. Call self.enable() or self.disable() to enable or disable the monitor. Call self.clear_recorded_data() to clear recorded data.

Example:

```python
import snntorch as snn
from snntorch.functional import probe

import torch
from torch import nn

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(8, 4)
        self.lif1 = snn.Leaky()
        self.fc2 = nn.Linear(4, 2)
        self.lif2 = snn.Leaky()

    def forward(self, x_seq: torch.Tensor):
        x_seq = self.fc1(x_seq)
        x_seq = self.lif1(x_seq)
        x_seq = self.fc2(x_seq)
        x_seq = self.lif2(x_seq)
        return x_seq

net = Net()

monitor = probe.GradInputMonitor(net, instance=snn.Leaky())

with torch.no_grad():
    y = net(torch.randn([1, 8]))
    print(f'monitor.records={monitor.records}"
    print(f'monitor[0]={monitor[0]}"
    print(f'monitor.monitored_layers={monitor.monitored_layers}"
    print(f'monitor['lif1']=monitor['lif1']"
```

Parameters

• net (nn.Module) – Network model (either wrapped in Sequential container or as a class)

• instance (Any or tuple) – Instance of modules to be monitored. If None, defaults to type(net)
• **function_on_grad_input** (*Callable, optional*) – Function that is applied to the monitored modules’ gradients

```python
create_hook(name)
```


Bases: BaseMonitor

A monitor to record the output gradient of each specific neuron layer (e.g. Leaky) in a network. All output gradient data is recorded in self.record as data type ‘list’. Call self.enable() or self.disable() to enable or disable the monitor. Call self.clear_recorded_data() to clear recorded data.

Example:

```python
import snntorch as snn
from snntorch.functional import probe
import torch
from torch import nn

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(8, 4)
        self.lif1 = snn.Leaky()
        self.fc2 = nn.Linear(4, 2)
        self.lif2 = snn.Leaky()

    def forward(self, x_seq: torch.Tensor):
        x_seq = self.fc1(x_seq)
        x_seq = self.lif1(x_seq)
        x_seq = self.fc2(x_seq)
        x_seq = self.lif2(x_seq)
        return x_seq

net = Net()

mtor = probe.GradOutputMonitor(net, instance=snn.Leaky())

with torch.no_grad():
    y = net(torch.rand([1, 8]))
    print(f'mtor.records={mtor.records}"
    print(f'mtor[0]={mtor[0]}""
    print(f'mtor.monitored_layers={mtor.monitored_layers}"
    print(f'mtor['lif1']=mtor['lif1']")
```

Parameters

- **net** (*nn.Module*) – Network model (either wrapped in Sequential container or as a class)
- **instance** (*Any or tuple*) – Instance of modules to be monitored. If None, defaults to type(net)
function_on_grad_output (Callable, optional) – Function that is applied to the monitored modules' gradients

class snntorch.functional.probe.InputMonitor

Parameters

- **net** (*nn.Module*) – Network model (either wrapped in Sequential container or as a class)

- **instance** (*Any or tuple*) – Instance of modules to be monitored. If None, defaults to type(net)

A monitor to record the input of each neuron layer (e.g. Leaky) in a network. All input data is recorded in self.record as data type "list". Call self.enable() or self.disable() to enable or disable the monitor. Call self.clear_recorded_data() to clear recorded data.

Example:

```python
import snntorch as snn
from snntorch.functional import probe
import torch
from torch import nn

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(8, 4)
        self.lif1 = snn.Leaky()
        self.fc2 = nn.Linear(4, 2)
        self.lif2 = snn.Leaky()

    def forward(self, x_seq: torch.Tensor):
        x_seq = self.fc1(x_seq)
        x_seq = self.lif1(x_seq)
        x_seq = self.fc2(x_seq)
        x_seq = self.lif2(x_seq)
        return x_seq

net = Net()
monitor = probe.InputMonitor(net, instance=snn.Leaky())

with torch.no_grad():
    y = net(torch.rand([1, 8]))
    print(f'monitor.records={monitor.records}')
    print(f'monitor[0]={monitor[0]}')
    print(f'monitor.monitored_layers={monitor.monitored_layers}''
    print(f'monitor[''lif1'']={monitor[''lif1'']}')
```

```
• **function_on_input** (*Callable, optional*) – Function that is applied to the monitored modules’ input

create_hook(*name*)


Bases: *BaseMonitor*

A monitor to record the output spikes of each specific neuron layer (e.g. Leaky) in a network. All output data is recorded in *self.record* as data type ‘list’. Call *self.enable()* or *self.disable()* to enable or disable the monitor. Call *self.clear_recorded_data()* to clear recorded data.

Example:

```python
import snntorch as snn
from snntorch.functional import probe
import torch
from torch import nn

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(8, 4)
        self.lif1 = snn.Leaky()
        self.fc2 = nn.Linear(4, 2)
        self.lif2 = snn.Leaky()

    def forward(self, x_seq: torch.Tensor):
        x_seq = self.fc1(x_seq)
        x_seq = self.lif1(x_seq)
        x_seq = self.fc2(x_seq)
        x_seq = self.lif2(x_seq)
        return x_seq

net = Net()
monitor = probe.OutputMonitor(net, instance=snntorch.Leaky())

with torch.no_grad():
    y = net(torch.rand([1, 8]))
    print(f'monitor.records={monitor.records}')
    print(f'monitor[0]={monitor[0]}')
    print(f'monitor.monitored_layers={monitor.monitored_layers}')
    print(f'monitor[''lif1'']={monitor[''lif1'']}')
```

Parameters

- **net** (*nn.Module*) – Network model (either wrapped in Sequential container or as a class)
- **instance** (*Any or tuple*) – Instance of modules to be monitored. If *None*, defaults to type(*net*)
• `function_on_output(Callable, optional) – Function that is applied to the monitored modules’ outputs

`create_hook(name)`

`snntorch.functional.probe.unpack_len1_tuple(x: tuple)`

1.11.6 snntorch.spikegen

`snntorch.spikegen` is a module that provides a variety of common spike generation and conversion methods, including spike-rate and latency coding.

How to use spikegen

In general, tensors containing non-spiking data can simply be passed into one of the functions in `snntorch.spikegen` to convert them into discrete spikes. There are a variety of methods to achieve this conversion. At present, `snntorch` supports:

• rate coding
• latency coding
• delta modulation

There are also options for converting targets into time-varying spikes.

`snntorch.spikegen.delta(data, threshold=0.1, padding=False, off_spike=False)`

Generate spike only when the difference between two subsequent time steps meets a threshold. Optionally include off_spikes for negative changes.

Example:

```python
a = torch.Tensor([1, 2, 2.9, 3, 3.9])
spikegen.delta(a, threshold=1)
>>> tensor([1., 1., 0., 1., 0.])

spikegen.delta(a, threshold=1, padding=True)
>>> tensor([0., 1., 0., 1., 0.])

b = torch.Tensor([1, 2, 0, 2, 2.9])
spikegen.delta(b, threshold=1, off_spike=True)
>>> tensor([1., 1., -1., 1., 0.])

spikegen.delta(b, threshold=1, padding=True, off_spike=True)
>>> tensor([0., 1., -1., 1., 0.])
```

Parameters

• `data (torch.Tensor) – Data tensor for a single batch of shape [num_steps x batch x input_size]`

• `threshold` – Input features with a change greater than the threshold across one timestep will generate a spike, defaults to 0.1

• `padding` – Used to change how the first time step of spikes are measured. If True, the first time step will be repeated with itself resulting in 0’s for the output spikes.
If False, the first time step will be padded with 0’s, defaults to False

Parameters

- **off_spike** (bool, optional) – If True, negative spikes for changes less than -threshold, defaults to False

snntorch.spikegen.from_one_hot(one_hot_label)

Convert one-hot encoding back into an integer

Example:

```python
one_hot_label = torch.tensor([[1., 0., 0., 0.],
                             [0., 1., 0., 0.],
                             [0., 0., 1., 0.],
                             [0., 0., 0., 1.]])
spikegen.from_one_hot(one_hot_label)
>>> tensor([0, 1, 2, 3])
```

Parameters

- **targets** (torch.Tensor) – one-hot label vector

Returns

- targets

Return type

- torch.Tensor

snntorch.spikegen.latency(data, num_steps=False, threshold=0.01, tau=1, first_spike_time=0, on_target=1, off_target=0, clip=False, normalize=False, linear=False, interpolate=False, bypass=False, epsilon=1e-07)

Latency encoding of input or target label data. Use input features to determine time-to-first spike. Expected inputs should be between 0 and 1.

Assume a LIF neuron model that charges up with time constant tau. Tensor dimensions use time first.

Example:

```python
a = torch.Tensor([[0.02, 0.5, 1]])
spikegen.latency(a, num_steps=5, normalize=True, linear=True)
>>> tensor([[0., 0., 1.],
           [0., 0., 0.],
           [0., 1., 0.],
           [0., 0., 0.],
           [1., 0., 0.]])
```

Parameters

- **data** (torch.Tensor) – Data tensor for a single batch of shape [batch x input_size]
- **num_steps** (int, optional) – Number of time steps. Explicitly needed if normalize=True, defaults to False (then changed to 1 if normalize=False)
- **threshold** (float, optional) – Input features below the threshold will fire at the final time step unless clip=True in which case they will not fire at all, defaults to 0.01
- **tau** (float, optional) – RC Time constant for LIF model used to calculate firing time, defaults to 1
- **first_spike_time** (int, optional) – Time to first spike, defaults to 0.
• **on_target** (*float, optional*) – Target at spike times, defaults to 1
• **off_target** (*float, optional*) – Target during refractory period, defaults to 0
• **clip** (*bool, optional*) – Option to remove spikes from features that fall below the threshold, defaults to False
• **normalize** (*bool, optional*) – Option to normalize the latency code such that the final spike(s) occur within `num_steps`, defaults to False
• **linear** (*bool, optional*) – Apply a linear latency code rather than the default logarithmic code, defaults to False
• **interpolate** (*bool, optional*) – Applies linear interpolation such that there is a gradually increasing target up to each spike, defaults to False
• **bypass** (*bool, optional*) – Used to block error messages that occur from either: i) spike times exceeding the bounds of `num_steps`, or ii) if `num_steps` is not specified, setting `bypass=True` allows the largest spike time to set `num_steps`. Defaults to False
• **epsilon** (*float, optional*) – A tiny positive value to avoid rounding errors when using `torch.arange`, defaults to `1e-7`

**Returns**
latency encoding spike train of features or labels

**Return type**
torch.Tensor

```python
snntorch.spikegen.latency_code(data, num_steps=False, threshold=0.01, tau=1, first_spike_time=0,
normalize=False, linear=False, epsilon=1e-07)
```

Latency encoding of input data. Convert input features or target labels to spike times. Assumes a LIF neuron model that charges up with time constant `tau` by default.

Example:
```python
da = torch.Tensor([0.02, 0.5, 1])
snntorch.spikegen.latency_code(a, num_steps=5, normalize=True, linear=True)
>>> (tensor([3.9200, 2.0000, 0.0000]), tensor([False, False, False]))
```

**Parameters**

• **data** (*torch.Tensor*) – Data tensor for a single batch of shape [batch x input_size]
• **num_steps** (*int, optional*) – Number of time steps. Explicitly needed if normalize=True, defaults to False (then changed to 1 if normalize=False)
• **threshold** (*float, optional*) – Input features below the threshold will fire at the final time step unless clip=True in which case they will not fire at all, defaults to `0.01`
• **tau** (*float, optional*) – RC Time constant for LIF model used to calculate firing time, defaults to 1
• **first_spike_time** (*int, optional*) – Time to first spike, defaults to 0.
• **normalize** (*bool, optional*) – Option to normalize the latency code such that the final spike(s) occur within `num_steps`, defaults to False
• **linear** (*bool, optional*) – Apply a linear latency code rather than the default logarithmic code, defaults to False
• **epsilon** (*float, optional*) – A tiny positive value to avoid rounding errors when using `torch.arange`, defaults to `1e-7`

**Returns**

latency encoding spike times of features

**Return type**

`torch.Tensor`

**Returns**

Tensor of Boolean values which correspond to the latency encoding elements that fall below the threshold. Used in `latency_conv` to clip saturated spikes.

**Return type**

`torch.Tensor`

---

`snntorch.spikegen.latency_code_linear(data, num_steps=False, threshold=0.01, tau=1, first_spike_time=0, normalize=False)`

Linear latency encoding of input data. Convert input features or target labels to spike times.

Example:

```python
a = torch.Tensor([0.02, 0.5, 1])
spikegen.latency_code(a, num_steps=5, normalize=True, linear=True)
>>> (tensor([3.9200, 2.0000, 0.0000]), tensor([False, False, False]))
```

**Parameters**

- **data** (*torch.Tensor*) – Data tensor for a single batch of shape `[batch x input_size]`
- **num_steps** (*int, optional*) – Number of time steps. Explicitly needed if `normalize=True`, defaults to `False` (then changed to `1` if `normalize=False`)
- **threshold** (*float, optional*) – Input features below the threshold will fire at the final time step, defaults to `0.01`
- **tau** (*float, optional*) – Linear time constant used to calculate firing time, defaults to `1`
- **first_spike_time** (*int, optional*) – Time to first spike, defaults to `0`
- **normalize** (*Bool, optional*) – Option to normalize the latency code such that the final spike(s) occur within `num_steps`, defaults to `False`

**Returns**

linear latency encoding spike times of features

**Return type**

`torch.Tensor`

---

`snntorch.spikegen.latency_code_log(data, num_steps=False, threshold=0.01, tau=1, first_spike_time=0, normalize=False, epsilon=1e-07)`

Logarithmic latency encoding of input data. Convert input features or target labels to spike times.

Example:

```python
a = torch.Tensor([0.02, 0.5, 1])
spikegen.latency_code(a, num_steps=5, normalize=True)
>>> (tensor([4.0000, 0.1166, 0.0580]), tensor([False, False, False]))
```

**Parameters**

---

1.11. License & Copyright
- **data** (torch.Tensor) – Data tensor for a single batch of shape [batch x input_size]
- **num_steps** (*int*, *optional*) – Number of time steps. Explicitly needed if `normalize=True`, defaults to False (then changed to 1 if `normalize=False`)
- **threshold** (*float*, *optional*) – Input features below the threshold will fire at the final time step, defaults to 0.01
- **tau** (*float*, *optional*) – Logarithmic time constant used to calculate firing time, defaults to 1
- **first_spike_time** (*int*, *optional*) – Time to first spike, defaults to 0.
- **normalize** (*Bool*, *optional*) – Option to normalize the latency code such that the final spike(s) occur within num_steps, defaults to False
- **epsilon** (*float*, *optional*) – A tiny positive value to avoid rounding errors when using torch.arange, defaults to 1e-7

**Returns**
logarithmic latency encoding spike times of features

**Return type**
torch.Tensor

`sntorch.spikegen.latency_interpolate(spike_time, num_steps, on_target=1, off_target=0)`

Apply linear interpolation to a tensor of target spike times to enable gradual increasing membrane. Each spike is assumed to occur from a separate neuron.

Example:

```python
a = torch.Tensor([0, 4])
spikegen.latency_interpolate(a, num_steps=5)
>>> tensor([[1.0000, 0.0000],
          [0.0000, 0.2500],
          [0.0000, 0.5000],
          [0.0000, 0.7500],
          [0.0000, 1.0000]])

spikegen.latency_interpolate(a, num_steps=5, on_target=1.25, off_target=0.25)
>>> tensor([[1.2500, 0.2500],
          [0.2500, 0.5000],
          [0.2500, 0.7500],
          [0.2500, 1.0000],
          [0.2500, 1.2500]])
```

**Parameters**

- **spike_time** – spike time targets in terms of steps
- **num_steps** (*int*, *optional*) – Number of time steps, defaults to False
- **on_target** (*float*, *optional*) – Target at spike times, defaults to 1
- **off_target** (*float*, *optional*) – Target during refractory period, defaults to 0

**Returns**
interpolated target of output neurons. Output tensor will use time-first dimensions.

**Return type**
torch.Tensor

Chapter 1. Introduction
snntorch.spikegen.rate(data, num_steps=False, gain=1, offset=0, first_spike_time=0, time_var_input=False)

Spike rate encoding of input data. Convert tensor into Poisson spike trains using the features as the mean of a binomial distribution. If num_steps is specified, then the data will be first repeated in the first dimension before rate encoding.

If data is time-varying, tensor dimensions use time first.

Example:

```python
# 100% chance of spike generation
a = torch.Tensor([1, 1, 1, 1])
spikegen.rate(a, num_steps=1)
>>> tensor([1., 1., 1., 1.])

# 0% chance of spike generation
b = torch.Tensor([0, 0, 0, 0])
spikegen.rate(b, num_steps=1)
>>> tensor([0., 0., 0., 0.])

# 50% chance of spike generation per time step
c = torch.Tensor([0.5, 0.5, 0.5, 0.5])
spikegen.rate(c, num_steps=1)
>>> tensor([0., 1., 0., 1.])

# Increasing num_steps will increase the length of the first dimension (time-first)
print(c.size())
>>> torch.Size([1, 4])
d = spikegen.rate(torch.Tensor([0.5, 0.5, 0.5, 0.5]), num_steps = 2)
print(d.size())
>>> torch.Size([2, 4])
```

Parameters

- **data** (torch.Tensor) – Data tensor for a single batch of shape [batch x input_size]
- **num_steps** (int, optional) – Number of time steps. Only specify if input data does not already have time dimension, defaults to False
- **gain** (float, optional) – Scale input features by the gain, defaults to 1
- **offset** (torch.optim, optional) – Shift input features by the offset, defaults to 0
- **first_spike_time** (int, optional) – Time to first spike, defaults to 0.
- **time_var_input** (bool, optional) – Set to True if input tensor is time-varying. Otherwise, first_spike_time!=0 will modify the wrong dimension. Defaults to False

Returns

rate encoding spike train of input features of shape [num_steps x batch x input_size]

Return type

torch.Tensor

snntorch.spikegen.rate_conv(data)

Convert tensor into Poisson spike trains using the features as the mean of a binomial distribution. Values outside the range of [0, 1] are clipped so they can be treated as probabilities.

Example:
# 100% chance of spike generation
a = torch.Tensor([1, 1, 1, 1])
spikegen.rate_conv(a)
>>> tensor([1., 1., 1., 1.])

# 0% chance of spike generation
b = torch.Tensor([0, 0, 0, 0])
spikegen.rate_conv(b)
>>> tensor([0., 0., 0., 0.])

# 50% chance of spike generation per time step
c = torch.Tensor([0.5, 0.5, 0.5, 0.5])
spikegen.rate_conv(c)
>>> tensor([0., 1., 0., 1.])

Parameters

- **data** (torch.Tensor) – Data tensor for a single batch of shape [batch x input_size]

Returns

- rate encoding spike train of input features of shape [num_steps x batch x input_size]

Return type

- torch.Tensor

snntorch.spikegen.rate_interpolate(spike_time, num_steps, on_target=1, off_target=0, epsilon=1e-07)

Apply linear interpolation to a tensor of target spike times to enable gradual increasing membrane.

Example:

```python
a = torch.Tensor([0, 4])
spikegen.rate_interpolate(a, num_steps=5)
>>> tensor([1.0000, 0.0000, 0.3333, 0.6667, 1.0000])
```

```python
spikegen.rate_interpolate(a, num_steps=5, on_target=1.25, off_target=0.25)
>>> tensor([1.2500, 0.2500, 0.5833, 0.9167, 1.2500])
```

Parameters

- **spike_time** – spike time targets in terms of steps
- **num_steps** (int, optional) – Number of time steps, defaults to False
- **on_target** (float, optional) – Target at spike times, defaults to 1
- **off_target** (float, optional) – Target during refractory period, defaults to 0
- **epsilon** (float, optional) – A tiny positive value to avoid rounding errors when using torch.arange, defaults to 1e-7

Returns

- interpolated target of output neurons. Output tensor will use time-first dimensions.

Return type

- torch.Tensor
snntorch.spikegen.target_rate_code(num_steps, first_spike_time=0, rate=1, firing_pattern='regular')

Rate coding a single output neuron of tensor of length num_steps containing spikes, and another tensor containing the spike times.

Example:

```python
spikegen.target_rate_code(num_steps=5, rate=1)
>>> (tensor([1., 1., 1., 1., 1.]), tensor([0, 1, 2, 3, 4]))

spikegen.target_rate_code(num_steps=5, first_spike_time=3, rate=1)
>>> (tensor([0., 0., 0., 1., 1.]), tensor([3, 4]))

spikegen.target_rate_code(num_steps=5, rate=0.3)
>>> (tensor([1., 0., 0., 1., 0.]), tensor([0, 3]))

spikegen.target_rate_code(num_steps=5, rate=0.3, firing_pattern="poisson")
>>> (tensor([0., 1., 0., 1., 0.]), tensor([1, 3]))
```

Parameters

- **num_steps** *(int, optional)* – Number of time steps, defaults to False
- **first_spike_time** *(int, optional)* – Time step for first spike to occur, defaults to 0
- **rate** *(float, optional)* – Firing frequency as a ratio, e.g., 1 enables firing at every step; 0.5 enables firing at 50% of steps, 0 means no firing, defaults to 1
- **firing_pattern** *(string, optional)* – Firing pattern of correct and incorrect classes. 'regular' enables periodic firing, 'uniform' samples spike times from a uniform distribution (duplicates are removed), 'poisson' samples from a binomial distribution at each step where each probability is the firing frequency, defaults to 'regular'

Returns

- **rate coded target of single neuron class of length num_steps**
- **torch.Tensor**

Returns

- **rate coded spike times in terms of steps**
- **torch.Tensor**

snntorch.spikegen.targets_convert(targets, num_classes, code='rate', num_steps=False, first_spike_time=0, correct_rate=1, incorrect_rate=0, on_target=1, off_target=0, firing_pattern='regular', interpolate=False, epsilon=1e-07, threshold=0.01, tau=1, clip=False, normalize=False, linear=False, bypass=False)

Spike encoding of targets. Expected input is a 1-D tensor with index of targets. If the output tensor is time-varying, the returned tensor will have time in the first dimension. If it is not time-varying, then the returned tensor will omit the time dimension and use batch first.

The following arguments will necessarily incur a time-varying output:
- **code** = 'latency', first_spike_time=0, correct_rate=1, or incorrect_rate=0

The target output may be applied to the internal state (e.g., membrane) of the neuron or to the spike. The following arguments will produce an output tensor that may sensibly be applied as a target to either the output spike or the membrane potential, as the output will consistently be either a 1 or 0:
on_target=1, off_target=0, and interpolate=False

If any of the above 3 conditions do not hold, then the target is better suited for the output membrane potential, as it will likely include values other than 1 and 0.

Example:

```python
a = torch.Tensor([4])

# rate-coding
# one-hot
spikegen.targets_convert(a, num_classes=5, code="rate")
>>> (tensor([[0., 0., 0., 0., 1.]]), )

# one-hot + time-first
spikegen.targets_convert(a, num_classes=5, code="rate", correct_rate=0.8, incorrect_rate=0.2, num_steps=5).size()
>>> torch.Size([5, 1, 5])
```

For more examples of rate-coding, see help(snntorch.spikegen(targets_rate)).

Parameters

- **targets** (torch.Tensor) – Target tensor for a single batch. The target should be a class index in the range [0, C-1] where C=number of classes.
- **num_classes** (int) – Number of outputs.
- **code** (string, optional) – Encoding scheme. Options of 'rate' or 'latency', defaults to 'rate'
- **num_steps** (int, optional) – Number of time steps, defaults to False
- **first_spike_time** (int, optional) – Time step for first spike to occur, defaults to 0
- **correct_rate** (float, optional) – Firing frequency of correct class as a ratio, e.g., 1 enables firing at every step; 0.5 enables firing at 50% of steps, 0 means no firing, defaults to 1
- **incorrect_rate** (float, optional) – Firing frequency of incorrect class(es), e.g., 1 enables firing at every step; 0.5 enables firing at 50% of steps, 0 means no firing, defaults to 0
- **on_target** (float, optional) – Target at spike times, defaults to 1
- **off_target** (float, optional) – Target during refractory period, defaults to 0
- **firing_pattern** (string, optional) – Firing pattern of correct and incorrect classes. 'regular' enables periodic firing, 'uniform' samples spike times from a uniform distributions (duplicates are removed), 'poisson' samples from a binomial distribution at each step where each probability is the firing frequency, defaults to 'regular'
- **interpolate** (Bool, optional) – Applies linear interpolation such that there is a gradually increasing target up to each spike, defaults to False
- **epsilon** (float, optional) – A tiny positive value to avoid rounding errors when using torch.arange, defaults to 1e-7
- **bypass** (bool, optional) – Used to block error messages that occur from either: i) spike times exceeding the bounds of num_steps, or ii) if num_steps is not specified, setting bypass=True allows the largest spike time to set num_steps. Defaults to False
Returns

spike coded target of output neurons. If targets are time-varying, the output tensor will use time-first dimensions. Otherwise, time is omitted from the tensor.

Return type
torch.Tensor

snntorch.spikegen.targets_latency(targets, num_classes, num_steps=False, first_spike_time=0, on_target=1, off_target=0, interpolate=False, threshold=0.01, tau=1, clip=False, normalize=False, linear=False, epsilon=1e-07, bypass=False)

Latency encoding of target labels. Use target labels to determine time-to-first spike. Expected input is index of correct class. The index is one-hot-encoded before being passed to spikegen.latency.

Assume a LIF neuron model that charges up with time constant tau. Tensor dimensions use time first.

Example:

```python
a = torch.Tensor([0, 3])
snntorch.targets_latency(a, num_classes=4, num_steps=5, normalize=True).size()
>>> torch.Size([5, 2, 4])

# time evolution of correct neuron class
snntorch.targets_latency(a, num_classes=4, num_steps=5, normalize=True)[..., 0, 0]
>>> tensor([1., 0., 0., 0., 0.])

# time evolution of incorrect neuron class
snntorch.targets_latency(a, num_classes=4, num_steps=5, normalize=True)[..., 0, 1]
>>> tensor([0., 0., 0., 0., 1.])

# correct class w/interpolation
snntorch.targets_latency(a, num_classes=4, num_steps=5, normalize=True, interpolate=True)[..., 0, 0]
>>> tensor([1., 0., 0., 0., 0.])

# incorrect class w/interpolation
snntorch.targets_latency(a, num_classes=4, num_steps=5, normalize=True, interpolate=True)[..., 0, 1]
>>> tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000])
```

Parameters

- **targets** (torch.Tensor) – Target tensor for a single batch. The target should be a class index in the range [0, C-1] where C=number of classes.
- **num_classes** (int) – Number of outputs.
- **num_steps** (int, optional) – Number of time steps. Explicitly needed if normalize=True, defaults to False (then changed to 1 if normalize=False)
- **first_spike_time** (int, optional) – Time to first spike, defaults to 0.
- **on_target** (float, optional) – Target at spike times, defaults to 1
- **off_target** (float, optional) – Target during refractory period, defaults to 0
- **interpolate** (Bool, optional) – Applies linear interpolation such that there is a gradually increasing target up to each spike, defaults to False
• **threshold** (*float, optional*) – Input features below the threshold will fire at the final time step unless `clip=True` in which case they will not fire at all, defaults to 0.01

• **tau** (*float, optional*) – RC Time constant for LIF model used to calculate firing time, defaults to 1

• **clip** (*bool, optional*) – Option to remove spikes from features that fall below the threshold, defaults to False

• **normalize** (*bool, optional*) – Option to normalize the latency code such that the final spike(s) occur within `num_steps`, defaults to False

• **linear** (*bool, optional*) – Apply a linear latency code rather than the default logarithmic code, defaults to False

• **bypass** (*bool, optional*) – Used to block error messages that occur from either: i) spike times exceeding the bounds of `num_steps`, or ii) if `num_steps` is not specified, setting `bypass=True` allows the largest spike time to set `num_steps`. Defaults to False

• **epsilon** (*float, optional*) – A tiny positive value to avoid rounding errors when using `torch.arange`, defaults to 1e-7

**Returns**

latency encoding spike train of features or labels

**Return type**

torch.Tensor

```python
snntorch.spikegen.targets_rate(targets, num_classes, num_steps=False, first_spike_time=0, correct_rate=1, incorrect_rate=0, on_target=1, off_target=0, firing_pattern='regular', interpolate=False, epsilon=1e-07)
```

Spike rate encoding of targets. Input tensor must be one-dimensional with target indexes. If the output tensor is time-varying, the returned tensor will have time in the first dimension. If it is not time-varying, the returned tensor will omit the time dimension and use batch first. If `first_spike_time!=0`, `correct_rate!=1`, or `incorrect_rate!=0`, the output tensor will be time-varying.

If `on_target=1`, `off_target=0`, and `interpolate=False`, then the target may sensibly be applied as a target for the output spike. If any of the above 3 conditions do not hold, then the target would be better suited for the output membrane potential.

**Example:**

```python
a = torch.Tensor([4])
# one-hot
spikegen.targets_rate(a, num_classes=5)
>>> (tensor([[0., 0., 0., 0., 1.]]), )

# first spike time delay, spike evolution over time
spikegen.targets_rate(a, num_classes=5, num_steps=5, first_spike_time=2).size()
>>> torch.Size([5, 1, 5])
spikegen.targets_rate(a, num_classes=5, num_steps=5, first_spike_time=2)[:, :, 4]
>>> (tensor([0., 0., 1., 1., 1.]))

# note: time has not been repeated because every time step would be identical where firsthand_spike_time defaults to 0
spikegen.targets_rate(a, num_classes=5, num_steps=5).size()
>>> torch.Size([1, 5])
```

(continues on next page)
# on/off targets - membrane evolution over time

```python
spikegen.targets_rate(a, num_classes=5, num_steps=5, first_spike_time=2, on_target=1.2, off_target=0.5)[..., 0, 4]
```

```python
>>> (tensor([0.5000, 0.5000, 1.2000, 1.2000, 1.2000]))
```

# correct rate at 25% + linear interpolation of membrane evolution

```python
spikegen.targets_rate(a, num_classes=5, num_steps=5, correct_rate=0.25, on_target=1.2, off_target=0.5, interpolate=True)[..., 0, 4]
```

```python
>>> tensor([1.2000, 0.5000, 0.7333, 0.9667, 1.2000])
```

Parameters

- **targets** (*torch.Tensor*) – Target tensor for a single batch. The target should be a class index in the range [0, C-1] where C=number of classes.
- **num_classes** (*int*) – Number of outputs.
- **num_steps** (*int, optional*) – Number of time steps, defaults to False
- **first_spike_time** (*int, optional*) – Time step for first spike to occur, defaults to 0
- **correct_rate** (*float, optional*) – Firing frequency of correct class as a ratio, e.g., 1 enables firing at every step; 0.5 enables firing at 50% of steps, 0 means no firing, defaults to 1
- **incorrect_rate** (*float, optional*) – Firing frequency of incorrect class(es), e.g., 1 enables firing at every step; 0.5 enables firing at 50% of steps, 0 means no firing, defaults to 0
- **on_target** (*float, optional*) – Target at spike times, defaults to 1
- **off_target** (*float, optional*) – Target during refractory period, defaults to 0
- **firing_pattern** (*string, optional*) – Firing pattern of correct and incorrect classes. 'regular' enables periodic firing, 'uniform' samples spike times from a uniform distributions (duplicates are removed), 'poisson' samples from a binomial distribution at each step where each probability is the firing frequency, defaults to 'regular'
- **interpolate** (*Bool, optional*) – Applies linear interpolation such that there is a gradually increasing target up to each spike, defaults to False
- **epsilon** (*float, optional*) – A tiny positive value to avoid rounding errors when using torch.arange, defaults to 1e-7

Returns

rate coded target of output neurons. If targets are time-varying, the output tensor will use time-first dimensions. Otherwise, time is omitted from the tensor.

Return type

torch.Tensor

**snntorch.spikegen.to_one_hot(targets, num_classes)**

One hot encoding of target labels.

Example:
targets = torch.tensor([0, 1, 2, 3])
spikegen.targets_to_spikes(targets, num_classes=4)

>>> tensor([[1., 0., 0., 0.],
          [0., 1., 0., 0.],
          [0., 0., 1., 0.],
          [0., 0., 0., 1.]])

Parameters

- **targets** (torch.Tensor) – Target tensor for a single batch
- **num_classes** (int) – Number of classes

Returns

one-hot encoding of targets of shape [batch x num_classes]

Return type

torch.Tensor

```python
snntorch.spikegen.to_one_hot_inverse(one_hot_targets)
```

Boolean inversion of a matrix of 1’s and 0’s. Used to merge the targets of correct and incorrect neuron classes in targets_rate.

Example:

```python
a = torch.Tensor([0, 0, 0, 0, 1])
spikegen.to_one_hot_inverse(a)
```

```
>>> tensor([[1., 1., 1., 1., 0.]])
```

1.11.7 snntorch.spikeplot

**snntorch.spikeplot** is deeply integrated with *matplotlib.pyplot* and *celluloid*. It serves to reduce the amount of boilerplate code required to generate a variety of animations and plots.

**class** snntorch.spikeplot.Camera(figure: Figure)

**Bases**: object

Make animations easier.

**animate** (*args, **kwargs) → ArtistAnimation

Animate the snapshots taken. Uses matplotlib.animation.ArtistAnimation

**Returns**: ArtistAnimation

**snap** () → List[Artist]

Capture current state of the figure.

**snntorch.spikeplot.alarm** (data, fig, ax, num_steps=False, interval=40, cmap='plasma')

Generate an animation by looping through the first dimension of a sample of spiking data. Time must be the first dimension of data.

Example:

```python
import snntorch.spikeplot as splt
import matplotlib.pyplot as plt

# spike_data contains 128 samples, each of 100 time steps in duration
```
print(spike_data.size())
>>> torch.Size([100, 128, 1, 28, 28])

# Index into a single sample from a minibatch
spike_data_sample = spike_data[:, 0, 0]
print(spike_data_sample.size())
>>> torch.Size([100, 28, 28])

# Plot
fig, ax = plt.subplots()
anim = splt.animator(spike_data_sample, fig, ax)
HTML(anim.to_html5_video())

# Save as a gif
anim.save("spike_mnist.gif")

Parameters

- **data** *(torch.Tensor)* – Data tensor for a single sample across time steps of shape [num_steps x input_size]
- **fig** *(matplotlib.figure.Figure)* – Top level container for all plot elements
- **ax** *(matplotlib.axes._subplots.AxesSubplot)* – Contains additional figure elements and sets the coordinate system. E.g.: fig, ax = plt.subplots(facecolor='w', figsize=(12, 7))
- **num_steps** *(int, optional)* – Number of time steps to plot. If not specified, the number of entries in the first dimension of data will automatically be used, defaults to False
- **interval** *(int, optional)* – Delay between frames in milliseconds, defaults to 40
- **cmap** *(string, optional)* – Color map, defaults to plasma

Returns

Animation to be displayed using matplotlib.pyplot.show()

Return type

FuncAnimation

`sntorch.spikeplot.raster(data, ax, **kwargs)`

Generate a raster plot using matplotlib.pyplot.scatter.

Example:

```python
import sntorch.spikeplot as splt
import matplotlib.pyplot as plt

# spike_data contains 128 samples, each of 100 time steps in duration
print(spike_data.size())
>>> torch.Size([100, 128, 1, 28, 28])

# Index into a single sample from a minibatch
spike_data_sample = spike_data[:, 0, 0]
print(spike_data_sample.size())
>>> torch.Size([100, 28, 28])
```
fig = plt.figure(facecolor="w", figsize=(10, 5))
ax = fig.add_subplot(111)

# s: size of scatter points; c: color of scatter points
splt.raster(spike_data_sample, ax, s=1.5, c="black")
plt.title("Input Layer")
plt.xlabel("Time step")
plt.ylabel("Neuron Number")
plt.show()

snntorch.spikeplot.spike_count(data, fig, ax, labels, num_steps=False, animate=False, interpolate=1, gridshader=True, interval=25, time_step=False)

Generate horizontal bar plot for a single forward pass. Options to animate also available.

Example:

```python
import snntorch.spikeplot as splt
import matplotlib.pyplot as plt
from IPython.display import HTML

num_steps = 25

# Use splt.spike_count to display behavior of output neurons for a single sample during feedforward

# spk_rec is a recording of output spikes across 25 time steps, using `batch_size=128`
print(spk_rec.size())
>>> torch.Size([25, 128, 10])

# We only need a single data sample
spk_results = torch.stack(spk_rec, dim=0)[:, :, :].to('cpu')
print(spk_results.size())
>>> torch.Size([25, 10])

fig, ax = plt.subplots(facecolor='w', figsize=(12, 7))
labels=['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

# Plot and save spike count histogram
splt.spike_count(spk_results, fig, ax, labels, num_steps = num_steps, time_step=1e-3)
plt.show()
plt.savefig('hist2.png', dpi=300, bbox_inches='tight')

# Animate and save spike count histogram
anim = splt.spike_count(spk_results, fig, ax, labels, animate=True, interpolate=5,
                        num_steps = num_steps, time_step=1e-3)
HTML(anim.to_html5_video())
anim.save("spike_bar.gif")
```

Parameters

- **data** (*torch.Tensor*) – Sample of spiking data across numerous time steps [num_steps x
num_outputs]

- **fig** *(matplotlib.figure.Figures)* – Top level container for all plot elements
- **ax** *(matplotlib.axes._subplots.AxesSubplot)* – Contains additional figure elements and sets the coordinate system. E.g., fig, ax = plt.subplots(facecolor='w', figsize=(12, 7))
- **labels** *(list)* – List of strings of the names of the output labels. E.g., for MNIST, labels = ['0', '1', '2', ..., '9']
- **num_steps** *(int, optional)* – Number of time steps to plot. If not specified, the number of entries in the first dimension of data will automatically be used, defaults to False
- **animate** *(Bool, optional)* – If True, return type matplotlib.animation.ArtistAnimation sequentially scanning across the range of time steps available in data. If False, display plot of the final step once all spikes have been counted, defaults to False
- **interpolate** *(int, optional)* – Can be increased to smooth the animation of the vertical time bar. The value passed is the interpolation factor: e.g., interpolate=1 results in no additional interpolation. e.g., interpolate=5 results in 4 additional frames for each time step, defaults to 1
- **gridshader** *(Bool, optional)* – Applies shading to figure background to distinguish output classes, defaults to True
- **interval** *(int, optional)* – Delay between frames in milliseconds, defaults to 25
- **time_step** *(int, optional)* – Duration of each time step in seconds. If False, time-axis will be in terms of num_steps. Else, time-axis is scaled by the argument passed, defaults to False

Returns
animation to be displayed using matplotlib.pyplot.show()

Return type
FuncAnimation (if animate is True)

**snntorch.spikeplot.traces** *(data, spk=None, dim=(3, 3), spk_height=5, titles=None)*

Plot an array of neuron traces (e.g., membrane potential or synaptic current). Optionally apply spikes to ride on the traces. *traces* was originally written by Friedemann Zenke.

Example:

```python
import snntorch.spikeplot as splt

# mem_rec contains the traces of 9 neuron membrane potentials across 100 time_steps in duration
mem_rec = torch.rand(100, 9)
print(mem_rec.size())
>>> torch.Size([100, 9])

# Plot
traces(mem_rec, dim=(3,3))
```

**Parameters**

- **data** *(torch.Tensor)* – Data tensor for neuron traces across time steps of shape [num_steps x num_neurons]
- **spk** *(torch.Tensor, optional)* – Data tensor for neuron traces across time steps of shape [num_steps x num_neurons], defaults to None
• **dim** *(tuple, optional)* – Dimensions of figure, defaults to *(3, 3)*
• **spk_height** *(float, optional)* – Height of spike to plot, defaults to 5
• **titles** *(list of strings, optional)* – Adds subplot titles, defaults to None

### 1.11.8 snntorch.spikevision

**Warning**: The spikevision module has been deprecated. To load neuromorphic datasets, we recommend using the Tonic project. For examples on how to use snnTorch together with Tonic, please refer to Tutorial 7 in the snnTorch Tutorial Series.

The **spikevision** module consists of neuromorphic datasets and common image transformations. It is the neuromorphic analog to torchvision.

spikevision contains the following neuromorphic datasets:

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Description</th>
<th>Author URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMNIST</td>
<td>A spiking version of the original frame-based MNIST dataset.</td>
<td>G. Orchard</td>
</tr>
<tr>
<td>DVSGesture</td>
<td>11 hand gestures recorded from 29 subjects under 3 illumination conditions using a DVS128.</td>
<td>IBM Research</td>
</tr>
<tr>
<td>SHD</td>
<td>Spikes in 700 input channels were generated using an artificial cochlea model listening to studio recordings of spoken digits from 0 to 9 in both German and English languages.</td>
<td>Zenke Lab</td>
</tr>
</tbody>
</table>

**Module Reference:**

**snntorch.spikevision.spikedata**

All datasets are subclasses of `torch.utils.data.Dataset` i.e., they have `__getitem__` and `__len__` methods implemented. Hence, they can all be passed to a `torch.utils.data.DataLoader` which can load multiple samples in parallel using `torch.multiprocessing` workers. For example:

```python
nmnist_data = spikevision.data.NMNIST('path/to/nmnist_root/')
data_loader = DataLoader(nmnist_data,
                         batch_size=4,
                         shuffle=True,
                         num_workers=--nThreads)
```

For further examples on each dataset and its use, please refer to the examples.
NMNIST

class snntorch.spikevision.spikedata.nmnist.NMNIST(root, train=True, transform=None, target_transform=None, download_and_create=True, num_steps=300, dt=1000)

NMNIST Dataset.

The Neuromorphic-MNIST (NMNIST) dataset is a spiking version of the original frame-based MNIST dataset. The downloaded and extracted dataset consists of the same 60000 training and 10000 testing samples as the MNIST dataset, and is captured at the same visual scale as the original MNIST dataset (28x28 pixels). For compatibility with the .hdf5 conversion process, this is reduced such that the number of samples for each class are balanced to the label with the minimum number of samples (training: 5421, test: 892).

Number of classes: 10
Number of train samples: 54210
Number of test samples: 8920
Dimensions: [num_steps x 2 x 32 x 32]
  • num_steps: time-dimension of event-based footage
  • 2: number of channels (on-spikes for luminance increasing; off-spikes for luminance decreasing)
  • 32x32: W x H spatial dimensions of event-based footage

For further reading, see:


Example:

```python
from snntorch.spikevision import spikedata

train_ds = spikedata.NMNIST("data/nmnist", train=True, num_steps=300, dt=1000)
test_ds = spikedata.NMNIST("data/nmnist", train=False, num_steps=300, dt=1000)

# by default, each time step is integrated over 1ms, or dt=1000 microseconds
# dt can be changed to integrate events over a varying number of time steps
# Note that num_steps should be scaled inversely by the same factor

train_ds = spikedata.NMNIST("data/nmnist", train=True, num_steps=150, dt=2000)
test_ds = spikedata.NMNIST("data/nmnist", train=False, num_steps=150, dt=2000)
```

The dataset can also be manually downloaded, extracted and placed into root which will allow the dataloader to bypass straight to the generation of a hdf5 file.

Direct Download Links:

Dropbox Train Set Link
Dropbox Test Set Link

Parameters

  • root (string) – Root directory of dataset where Train.zip and Test.zip exist.
• **train** *(bool, optional)* – If True, creates dataset from Train.zip, otherwise from Test.zip

• **transform** *(callable, optional)* – A function/transform that takes in a PIL image and returns a transforms version. By default, a pre-defined set of transforms are applied to all samples to convert them into a time-first tensor with correct orientation.

• **target_transform** *(callable, optional)* – A function/transform that takes in the target and transforms it.

• **download_and_create** *(bool, optional)* – If True, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.

• **num_steps** *(int, optional)* – Number of time steps, defaults to 300

• **dt** *(int, optional)* – Number of time stamps integrated in microseconds, defaults to 1000

Dataloader adapted from torchneuromorphic originally by Emre Neftci and Clemens Schaefer.

The dataset is released under the Creative Commons Attribution-ShareAlike 4.0 license. All rights remain with the original authors.

**DVSGesture**

class snntorch.spikevision.spikedata.dvs_gesture.DVSGesture(root, train=True, transform=None, target_transform=None, download_and_create=True, num_steps=None, dt=1000, ds=None, return_meta=False, time_shuffle=False)

DVS Gesture Dataset.

The data was recorded using a DVS128. The dataset contains 11 hand gestures from 29 subjects under 3 illumination conditions.

**Number of classes:** 11

**Number of train samples:** 1176

**Number of test samples:** 288

**Dimensions:** [num_steps x 2 x 128 x 128]

• **num_steps**: time-dimension of event-based footage

• **2**: number of channels (on-spikes for luminance increasing; off-spikes for luminance decreasing)

• **128x128**: W x H spatial dimensions of event-based footage

For further reading, see:


Example:
```python
from snntorch.spikevision import spikedata

train_ds = spikedata.DVSGesture("data/dvsgesture", train=True, num_steps=500, dt=1000)
test_ds = spikedata.DVSGesture("data/dvsgesture", train=False, num_steps=1800, dt=1000)

# by default, each time step is integrated over 1ms, or dt=1000 microseconds
# dt can be changed to integrate events over a varying number of time steps
# Note that num_steps should be scaled inversely by the same factor

train_ds = spikedata.DVSGesture("data/dvsgesture", train=True, num_steps=250, dt=2000)
test_ds = spikedata.DVSGesture("data/dvsgesture", train=False, num_steps=900, dt=2000)
```

The dataset can also be manually downloaded, extracted and placed into `root` which will allow the dataloader to bypass straight to the generation of a hdf5 file.

**Direct Download Links:**
- IBM Box Link
- Dropbox Link

**Parameters**

- `root` *(string)* – Root directory of dataset.
- `train` *(bool, optional)* – If True, creates dataset from training set of dvsgesture, otherwise test set.
- `transform` *(callable, optional)* – A function/transform that takes in a PIL image and returns a transforms version. By default, a pre-defined set of transforms are applied to all samples to convert them into a time-first tensor with correct orientation.
- `target_transform` *(callable, optional)* – A function/transform that takes in the target and transforms it.
- `download_and_create` *(bool, optional)* – If True, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- `num_steps` *(int, optional)* – Number of time steps, defaults to 500 for train set, or 1800 for test set
- `dt` *(int, optional)* – The number of time stamps integrated in microseconds, defaults to 1000
- `ds` *(int, optional)* – Rescaling factor, defaults to 1.

**Return_meta**
Option to return metadata, defaults to False

**Time_shuffle**
Option to randomize start time of dataset, defaults to False

Dataloader adapted from `torchneuromorphic` originally by Emre Neftci and Clemens Schaefer.
The dataset is released under a Creative Commons Attribution 4.0 license. All rights remain with the original authors.

**SHD**

**class snntorch.spikevision.spikedata.shd.SHD**(root=True, transform=None, target_transform=None, download_and_create=True, num_steps=1000, ds=1, dt=1000)

Spiking Heidelberg Digits Dataset.

Spikes in 700 input channels were generated using an artificial cochlea model listening to studio recordings of spoken digits from 0 to 9 in both German and English languages.

**Number of classes**: 20

**Number of train samples**: 8156

**Number of test samples**: 2264

**Dimensions**: [num_steps x 700]

- **num_steps**: time-dimension of audio channels
- **700**: number of channels in cochlea model

For further reading, see:


Example:

```python
from snntorch.spikevision import spikedata

train_ds = spikedata.SHD("data/shd", train=True)
test_ds = spikedata.SHD("data/shd", train=False)

# by default, each time step is integrated over 1ms, or dt=1000 microseconds
# dt can be changed to integrate events over a varying number of time steps
# Note that num_steps should be scaled inversely by the same factor

train_ds = spikedata.SHD("data/shd", train=True, num_steps=500, dt=2000)
test_ds = spikedata.SHD("data/shd", train=False, num_steps=500, dt=2000)
```

The dataset can also be manually downloaded, extracted and placed into `root` which will allow the dataloader to bypass straight to the generation of a hdf5 file.

**Direct Download Links:**

- CompNeuro Train Set Link
- CompNeuro Test Set Link

**Parameters**

- **root** *(string)* – Root directory of dataset.
- **train** *(bool, optional)* – If True, creates dataset from training set of dvsgesture, otherwise test set.
• **transform** (*callable, optional*) – A function/transform that takes in a PIL image and returns a transforms version. By default, a pre-defined set of transforms are applied to all samples to convert them into a time-first tensor with correct orientation.

• **target_transform** (*callable, optional*) – A function/transform that takes in the target and transforms it.

• **download_and_create** (*bool, optional*) – If True, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.

• **num_steps** (*int, optional*) – Number of time steps, defaults to **1000**

• **dt** (*int, optional*) – The number of time stamps integrated in microseconds, defaults to **1000**

• **ds** (*int, optional*) – Rescaling factor, defaults to 1.

Dataloader adapted from **torchneuromorphic** originally by Emre Neftci.

The dataset is released under a Creative Commons Attribution 4.0 International License. All rights remain with the original authors.

### 1.11.9 **snntorch.surrogate**

By default, PyTorch’s autodifferentiation tools are unable to calculate the analytical derivative of the spiking neuron graph. The discrete nature of spikes makes it difficult for **torch.autograd** to calculate a gradient that facilitates learning. **snntorch** overrides the default gradient by using **snntorch.LIF.Heaviside**.

Alternative gradients are also available in the **snntorch.surrogate** module. These represent either approximations of the backward pass or probabilistic models of firing as a function of the membrane potential.

At present, the surrogate gradient functions available include:

• Sigmoid
• Fast Sigmoid
• ATan
• Straight Through Estimator
• Triangular
• SpikeRateEscape

amongst several other options.

For further reading, see:

How to use surrogate

The surrogate gradient must be passed as the spike_grad argument to the neuron model. If spike_grad is left unspecified, it defaults to `snntorch.neurons.Heaviside`. In the following example, we apply the fast sigmoid surrogate to `snntorch.Synaptic`.

Example:

```python
import snntorch as snn
from snntorch import surrogate
import torch
import torch.nn as nn

alpha = 0.9
beta = 0.85

# Initialize surrogate gradient
spike_grad1 = surrogate.fast_sigmoid()  # passes default parameters from a closure
spike_grad2 = surrogate.FastSigmoid.apply  # passes default parameters, equivalent to above
spike_grad3 = surrogate.fast_sigmoid(slope=50)  # custom parameters from a closure

# Define Network
class Net(nn.Module):
    def __init__(self):
        super().__init__()

    # Initialize layers, specify the `spike_grad` argument
    self.fc1 = nn.Linear(num_inputs, num_hidden)
    self.lif1 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=spike_grad1)
    self.fc2 = nn.Linear(num_hidden, num_outputs)
    self.lif2 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=spike_grad3)

    def forward(self, x, syn1, mem1, spk1, syn2, mem2):
        cur1 = self.fc1(x)
        spk1, syn1, mem1 = self.lif1(cur1, syn1, mem1)
        cur2 = self.fc2(spk1)
        spk2, syn2, mem2 = self.lif2(cur2, syn2, mem2)
        return syn1, mem1, spk1, syn2, mem2, spk2

net = Net().to(device)
```

class `snntorch.surrogate.ATan(*args, **kwargs)`

Bases: Function

Surrogate gradient of the Heaviside step function.

Forward pass: Heaviside step function shifted.

\[
S = \begin{cases} 
1 & \text{if } U \geq U_{\text{thr}} \\
0 & \text{if } U < U_{\text{thr}}
\end{cases}
\]

Backward pass: Gradient of shifted arc-tan function.
\[
S = \frac{1}{1 + \left(\frac{U}{\alpha}\right)^2}
\]

\[
S = \frac{1}{1 + \left(\frac{U}{\alpha}\right)^2}
\]

alpha defaults to 2, and can be modified by calling surrogate.atan(alpha=2).

Adapted from:


static backward(ctx, grad_output)

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by as many outputs as the forward() returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to forward(). Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute ctx.needs_input_grad as a tuple of booleans representing whether each input needs gradient. E.g., backward() will have ctx.needs_input_grad[0] = True if the first input to forward() needs gradient computed w.r.t. the output.

static forward(ctx, input_, alpha=2.0)

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on ctx (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with ctx.save_for_backward() if they are intended to be used in backward (equivalently, vjp) or ctx.save_for_forward() if they are intended to be used for in jvp.

class snntorch.surrogate.FastSigmoid(*args, **kwargs)

Bases: Function

Surrogate gradient of the Heaviside step function.

Forward pass: Heaviside step function shifted.

\[
S = \begin{cases} 
1 & \text{if } U \geq U_{\text{thr}} \\
0 & \text{if } U < U_{\text{thr}} 
\end{cases}
\]

Backward pass: Gradient of fast sigmoid function.
\[
S = \frac{U}{1 + k|U|} \\
S = \frac{1}{(1 + k|U|)^2}
\]

\(k\) defaults to 25, and can be modified by calling `surrogate.fast_sigmoid(slope=25)`.

Adapted from:

**static backward**(\(ctx, grad\_output\))

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context \(ctx\) as the first argument, followed by as many outputs as the `forward()` returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to `forward()`. Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx.needs_input_grad` as a tuple of booleans representing whether each input needs gradient. E.g., `backward()` will have `ctx.needs_input_grad[0] = True` if the first input to `forward()` needs gradient computed w.r.t. the output.

**static forward**(\(ctx, input_\), slope=25)

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context \(ctx\) as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on \(ctx\) (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with `ctx.save_for_backward()` if they are intended to be used in `backward` (equivalently, vjp) or `ctx.save_for_forward()` if they are intended to be used for in jvp.

```python
snntorch.surrogate.LSO(slope=0.1)
```

Leaky spike operator gradient enclosed with a parameterized slope.

**class** `snntorch.surrogate.LeakySpikeOperator(*args, **kwargs)`

Bases: `Function`

Surrogate gradient of the Heaviside step function.

**Forward pass:** Spike operator function.

\[
S = \begin{cases} 
\frac{U(t)}{U} & \text{if } U > U_{\text{thr}} \\
0 & \text{if } U < U_{\text{thr}} 
\end{cases}
\]

**Backward pass:** Leaky gradient of spike operator, where the subthreshold gradient is treated as a small constant slope.
\[
S = \begin{cases} 
\frac{U(t)}{U} & \text{if } U > U_{\text{thr}} \\
U(t) & \text{if } U = U_{\text{thr}} \\
kU & \text{if } U < U_{\text{thr}} 
\end{cases}
\]

\[S = \begin{cases} 
1 & \text{if } U > U_{\text{thr}} \\
k & \text{if } U < U_{\text{thr}} 
\end{cases}\]

k defaults to 0.1, and can be modified by calling `surrogate.LSO(slope=0.1)`. The gradient is identical to that of a threshold-shifted Leaky ReLU.

**static backward** *(ctx, grad_output)*

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by as many outputs as the `forward()` returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to `forward()`. Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx.needs_input_grad` as a tuple of booleans representing whether each input needs gradient. E.g., `backward()` will have `ctx.needs_input_grad[0] = True` if the first input to `forward()` needs gradient computed w.r.t. the output.

**static forward** *(ctx, input_, slope=0.1)*

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on `ctx` (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with `ctx.save_for_backward()` if they are intended to be used in `backward` (equivalently, vjp) or `ctx.save_for_forward()` if they are intended to be used for in jvp.

snntorch.surrogate.SFS(*slope*=25, *B*=1)

SparseFastSigmoid surrogate gradient enclosed with a parameterized slope and sparsity threshold.

snntorch.surrogate.SSO(*mean*=0, *variance*=0.2)

Stochastic spike operator gradient enclosed with a parameterized mean and variance.

**class** snntorch.surrogate.Sigmoid(*args, **kwargs)*

Bases: Function

Surrogate gradient of the Heaviside step function.

**Forward pass:** Heaviside step function shifted.
**Backward pass:** Gradient of sigmoid function.

\[
S = \frac{1}{1 + \exp(-kU)}
\]

\[
\frac{S}{U} = \frac{k \exp(-kU)}{[\exp(-kU) + 1]^2}
\]

\(k\) defaults to 25, and can be modified by calling `surrogate.sigmoid(slope=25)`.

Adapted from:


**static backward** (ctx, grad_output)

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by as many outputs as the `forward()` returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to `forward()`. Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx.needs_input_grad` as a tuple of booleans representing whether each input needs gradient. E.g., `backward()` will have `ctx.needs_input_grad[0] = True` if the first input to `forward()` needs gradient computed w.r.t. the output.

**static forward** (ctx, input_, slope=25)

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on ctx (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with `ctx.save_for_backward()` if they are intended to be used in `backward` (equivalently, vjp) or `ctx.save_for_forward()` if they are intended to be used for in vjp.

**class** snntorch.surrogate.SparseFastSigmoid(*args, **kwargs)

Bases: Function

Surrogate gradient of the Heaviside step function.

**Forward pass:** Heaviside step function shifted.

\[
S = \begin{cases} 
1 & \text{if } U < U_{\text{thr}} \\
0 & \text{if } U \geq U_{\text{thr}} 
\end{cases}
\]

**Backward pass:** Gradient of fast sigmoid function clipped below B.
$$S(U) = \frac{U}{1 + k|U|} H(U - B)$$

$$\frac{S}{U} = \begin{cases} 
1 & \text{if } U > B \\
\frac{1}{(1 + k|U|)^2} & \text{otherwise}
\end{cases}$$

$k$ defaults to 25, and can be modified by calling `surrogate.SFS(slope=25)`. $B$ defaults to 1, and can be modified by calling `surrogate.SFS(B=1)`.

Adapted from:


**static backward** (ctx, grad_output)

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by as many outputs as the `forward()` returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to `forward()`. Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx.needs_input_grad` as a tuple of booleans representing whether each input needs gradient. E.g., `backward()` will have `ctx.needs_input_grad[0] = True` if the first input to `forward()` needs gradient computed w.r.t. the output.

**static forward** (ctx, input_, slope=25, B=1)

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on `ctx` (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with `ctx.save_for_backward()` if they are intended to be used in `backward` (equivalently, `vjp`) or `ctx.save_for_forward()` if they are intended to be used for in `jvp`.

**class** `snntorch.surrogate.SpikeRateEscape(*args, **kwargs)`

Bases: `Function`

Surrogate gradient of the Heaviside step function.

**Forward pass:** Heaviside step function shifted.

$$S = \begin{cases} 
1 & \text{if } U \geq U_{\text{thr}} \\
0 & \text{if } U < U_{\text{thr}}
\end{cases}$$

**Backward pass:** Gradient of Boltzmann Distribution.
\[
\frac{S}{U} = k \exp(-|U - 1|)
\]

defaults to 1, and can be modified by calling `surrogate.spike_rate_escape(beta=1)`. \(k\) defaults to 25, and can be modified by calling `surrogate.spike_rate_escape(slope=25)`.

Adapted from:


### backward(grad_output)

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by as many outputs as the `forward()` returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to `forward()`. Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx.needs_input_grad` as a tuple of booleans representing whether each input needs gradient. E.g., `backward()` will have `ctx.needs_input_grad[0] = True` if the first input to `forward()` needs gradient computated w.r.t. the output.

### static forward(ctx, input_, beta=1, slope=25)

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on `ctx` (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with `ctx.save_for_backward()` if they are intended to be used in `backward` (equivalently, vjp) or `ctx.save_for_forward()` if they are intended to be used for in `jvp`.

### class snntorch.surrogate.StochasticSpikeOperator(*args, **kwargs)

Bases: `Function`

Surrogate gradient of the Heaviside step function.

**Forward pass:** Spike operator function.

\[
S = \begin{cases} 
\frac{U(t)}{U} & \text{if } U > U_{thr} \\
0 & \text{if } U < U_{thr}
\end{cases}
\]

**Backward pass:** Gradient of spike operator, where the subthreshold gradient is sampled from uniformly distributed noise on the interval \((\sim [-0.5, 0.5] +)^2\), where \(\mu\) is the mean and \(\sigma^2\) is the variance.
The above defaults set the gradient to the following expression:

$$S = \begin{cases} 1 & \text{if } U \geq U_{\text{thr}} \\ (\sim [-0.1, 0.1])^2 & \text{if } U < U_{\text{thr}} \end{cases}$$

2 defaults to 0.2, and can be modified by calling `surrogate.SS0(variance=0.5)`.

The above defaults set the gradient to the following expression:

$$S = \begin{cases} 1 & \text{if } U \geq U_{\text{thr}} \\ (\sim [-0.1, 0.1])^2 & \text{if } U < U_{\text{thr}} \end{cases}$$

### static backward

```python
define a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).
```

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by as many outputs as the `forward()` returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to `forward()`. Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx.needs_input_grad` as a tuple of booleans representing whether each input needs gradient. E.g., `backward()` will have `ctx.needs_input_grad[0] = True` if the first input to `forward()` needs gradient computed w.r.t. the output.

### static forward

```python
performs the operation.
```

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on `ctx` (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with `ctx.save_for_backward()` if they are intended to be used in `backward` (equivalently, vjp) or `ctx.save_for_forward()` if they are intended to be used for in vjp.

### class snntorch.surrogateStraightThroughEstimator(*args, **kwargs)

Bases: `Function`

Straight Through Estimator.

**Forward pass:** Heaviside step function shifted.
**Backward pass:** Gradient of fast sigmoid function.

\[
\frac{S}{U} = 1
\]

`static backward(ctx, grad_output)`

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by as many outputs as the `forward()` returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to `forward()`. Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx.needs_input_grad` as a tuple of booleans representing whether each input needs gradient. E.g., `backward()` will have `ctx.needs_input_grad[0] = True` if the first input to `forward()` needs gradient computed w.r.t. the output.

`static forward(ctx, input_)`

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on `ctx` (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with `ctx.save_for_backward()` if they are intended to be used in backward (equivalently, vjp) or `ctx.save_for_forward()` if they are intended to be used for in jvp.

class snntorch.surrogate.Triangular(*args, **kwargs)

Bases: Function

Triangular Surrogate Gradient.

**Forward pass:** Heaviside step function shifted.

\[
S = \begin{cases} 
1 & \text{if } U \geq U_{\text{thr}} \\
0 & \text{if } U < U_{\text{thr}} 
\end{cases}
\]

**Backward pass:** Gradient of the triangular function.

\[
\frac{S}{U} = \begin{cases} 
U_{\text{thr}} & \text{if } U < U_{\text{thr}} \\
-U_{\text{thr}} & \text{if } U \geq U_{\text{thr}} 
\end{cases}
\]
static backward \((ctx, \text{grad\_output})\)

Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context \(ctx\) as the first argument, followed by as many outputs as the \(\text{forward()}\) returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to \(\text{forward()}\). Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute \(ctx.\text{needs\_input\_grad}\) as a tuple of booleans representing whether each input needs gradient. E.g., \(\text{backward()}\) will have \(ctx.\text{needs\_input\_grad}[0] = \text{True}\) if the first input to \(\text{forward()}\) needs gradient computed w.r.t. the output.

static forward \((ctx, input\_\_, \text{threshold}=1)\)

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context \(ctx\) as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on \(ctx\) (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with \(ctx.\text{save\_for\_backward}()\) if they are intended to be used in \(\text{backward}\) (equivalently, vjp) or \(ctx.\text{save\_for\_forward}()\) if they are intended to be used for in jvp.

\[\text{snntorch.surrogate.atan}(alpha=2.0)\]

ArcTan surrogate gradient enclosed with a parameterized slope.

\[\text{snntorch.surrogate.fast\_sigmoid}(slope=25)\]

FastSigmoid surrogate gradient enclosed with a parameterized slope.

\[\text{snntorch.surrogate.heaviside()}\]

Heaviside surrogate gradient wrapper.

\[\text{snntorch.surrogate.sigmoid}(slope=25)\]

Sigmoid surrogate gradient enclosed with a parameterized slope.

\[\text{snntorch.surrogate.spike\_rate\_escape}(beta=1, slope=25)\]

SpikeRateEscape surrogate gradient enclosed with a parameterized slope.

\[\text{snntorch.surrogate.straight\_through\_estimator}()\]

Straight Through Estimator surrogate gradient enclosed with a parameterized slope.

\[\text{snntorch.surrogate.triangular}()\]

Triangular surrogate gradient enclosed with a parameterized threshold.
1.11.10 snntorch.utils

`snntorch.utils` contains a handful of utility functions for handling datasets.

**snntorch.utils.data_subset(dataset, subset, idx=0)**

Partition the dataset by a factor of 1/subset without removing access to data and target attributes.

Example:

```python
from snntorch import utils
from torchvision import datasets

data_path = "path/to/data"
subset = 10

# Download MNIST training set
mnist_train = datasets.MNIST(data_path, train=True, download=True)
print(len(mnist_train))
>>> 60000

# Reduce size of MNIST training set
utils.data_subset(mnist_train, subset)
print(len(mnist_train))
>>> 6000
```

**Parameters**

- `dataset` (torchvision dataset) – Dataset
- `subset` (int) – Factor to reduce dataset by
- `idx` (int, optional) – Which subset of the train and test sets to index into, defaults to 0

**Returns**

Partitioned dataset

**Return type**

list of torch.utils.data

**snntorch.utils.reset(net)**

Check for the types of LIF neurons contained in net. Reset their hidden parameters to zero and detach them from the current computation graph.

**snntorch.utils.valid_split(ds_train, ds_val, split, seed=0)**

Randomly split a dataset into non-overlapping new datasets of given lengths. Optionally fix the generator for reproducible results. Operates similarly to random_split from torch.utils.data.dataset but retains data and target attributes.

Example

```python
from snntorch import utils
from torchvision import datasets

data_path = "path/to/data"
val_split = 0.1

# Download MNIST training set into mnist_val and mnist_train
```
mnist_train = datasets.MNIST(data_path, train=True, download=True)
mnist_val = datasets.MNIST(data_path, train=True, download=True)

print(len(mnist_train))
>>> 60000

print(len(mnist_val))
>>> 60000

# Validation split
mnist_train, mnist_val = utils.valid_split(mnist_train, mnist_val, val_split)

print(len(mnist_train))
>>> 54000

print(len(mnist_val))
>>> 6000

Parameters

- **ds_train** (*torchvision dataset*) – Training set
- **ds_val** (*torchvision dataset*) – Validation set
- **split** (*Float*) – Proportion of samples assigned to the validation set from the training set
- **seed** (*int, optional*) – Fix to generate reproducible results, defaults to 0

Returns

Randomly split train and validation sets

Return type

list of torch.utils.data

1.11.11 Quickstart

Tutorial written by Jason K. Eshraghian ([www.jasoneshraghian.com](http://www.jasoneshraghian.com))

For a comprehensive overview on how SNNs work, and what is going on under the hood, then you might be interested in the snnTorch tutorial series available here. The snnTorch tutorial series is based on the following paper. If you find these resources or code useful in your work, please consider citing the following source:


Note:

This tutorial is a static non-editable version. Interactive, editable versions are available via the following links:

- Google Colab
- Local Notebook (download via GitHub)
pip install snntorch

```python
import torch, torch.nn as nn
import snntorch as snn
```

## DataLoading

Define variables for dataloading.

```python
batch_size = 128
data_path='/data/mnist'
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
```

Load MNIST dataset.

```python
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Define a transform
transform = transforms.Compose([transforms.Resize((28, 28)), transforms.Grayscale(), transforms.ToTensor(), transforms.Normalize((0,), (1,))])

mnist_train = datasets.MNIST(data_path, train=True, download=True, transform=transform)
mnist_test = datasets.MNIST(data_path, train=False, download=True, transform=transform)

# Create DataLoaders
train_loader = DataLoader(mnist_train, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(mnist_test, batch_size=batch_size, shuffle=True)
```

## Define Network with snnTorch.

- `snn.Leaky()` instantiates a simple leaky integrate-and-fire neuron.
- `spike_grad` optionally defines the surrogate gradient. If left undefined, the relevant gradient term is simply set to the output spike itself (1/0) by default.

The problem with `nn.Sequential` is that each hidden layer can only pass one tensor to subsequent layers, whereas most spiking neurons return their spikes and hidden state(s). To handle this:

- `init_hidden` initializes the hidden states (e.g., membrane potential) as instance variables to be processed in the background.

The final layer is not bound by this constraint, and can return multiple tensors:

- `output=True` enables the final layer to return the hidden state in addition to the spike.

```python
from snntorch import surrogate
beta = 0.9  # neuron decay rate
spike_grad = surrogate.fast_sigmoid()
```

(continues on next page)
# Initialize Network

```python
net = nn.Sequential(nn.Conv2d(1, 8, 5),
                    nn.MaxPool2d(2),
                    snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True),
                    nn.Conv2d(8, 16, 5),
                    nn.MaxPool2d(2),
                    snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True),
                    nn.Flatten(),
                    nn.Linear(16*4*4, 10),
                    snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True, output=True)).to(device)
```

Refer to the snnTorch documentation to see more neuron types and surrogate gradient options.

**Define the Forward Pass**

Now define the forward pass over multiple time steps of simulation.

```python
from snntorch import utils
def forward_pass(net, data, num_steps):
    spk_rec = []
    utils.reset(net)  # resets hidden states for all LIF neurons in net
    for step in range(num_steps):
        spk_out, mem_out = net(data)
        spk_rec.append(spk_out)
    return torch.stack(spk_rec)
```

Define the optimizer and loss function. Here, we use the MSE Count Loss, which counts up the total number of output spikes at the end of the simulation run. The correct class has a target firing rate of 80% of all time steps, and incorrect classes are set to 20%.

```python
import snntorch.functional as SF
optimizer = torch.optim.Adam(net.parameters(), lr=2e-3, betas=(0.9, 0.999))
loss_fn = SF.mse_count_loss(correct_rate=0.8, incorrect_rate=0.2)
```

Objective functions do not have to be applied to the spike count. They may be applied to the membrane potential (hidden state), or to spike-timing targets instead of rate-based methods. A non-exhaustive list of objective functions available include:

**Apply the objective directly to spikes:**

- MSE Spike Count Loss: `mse_count_loss()`
- Cross Entropy Spike Count Loss: `ce_count_loss()`
- Cross Entropy Spike Rate Loss: `ce_rate_loss()`

**Apply the objective to the hidden state:**
• Cross Entropy Maximum Membrane Potential Loss: `ce_max_membrane_loss()
• MSE Membrane Potential Loss: `mse_membrane_loss()

For alternative objective functions, refer to the `snntorch.functional documentation here.

Training Loop

Now for the training loop. The predicted class will be set to the neuron with the highest firing rate, i.e., a rate-coded output. We will just measure accuracy on the training set. This training loop follows the same syntax as with PyTorch.

```python
num_epochs = 1
num_steps = 25  # run for 25 time steps

loss_hist = []
acc_hist = []

# training loop
for epoch in range(num_epochs):
    for i, (data, targets) in enumerate(iter(train_loader)):
        data = data.to(device)
        targets = targets.to(device)

        net.train()
        spk_rec = forward_pass(net, data, num_steps)
        loss_val = loss_fn(spk_rec, targets)

        # Gradient calculation + weight update
        optimizer.zero_grad()
        loss_val.backward()
        optimizer.step()

        # Store loss history for future plotting
        loss_hist.append(loss_val.item())

        # print every 25 iterations
        if i % 25 == 0:
            print(f"Epoch {epoch}, Iteration {i} 
Train Loss: {loss_val.item():.2f}"
)

        # check accuracy on a single batch
        acc = SF.accuracy_rate(spk_rec, targets)
        acc_hist.append(acc)
        print(f"Accuracy: {acc * 100:.2f}%\n"
)

    # uncomment for faster termination
    # if i == 150:
    #     break
```
Automating Backprop

Alternatively, we can automate the backprop through time training process using the BPTT method available in \texttt{snntorch.backprop}. All model updates take place within the \texttt{backprop.BPTT} function call. The specified number of steps in \texttt{num_steps} will be simulated just as before.

```python
from snntorch import backprop

num_epochs = 3

# training loop
for epoch in range(num_epochs):
    avg_loss = backprop.BPTT(net, train_loader, num_steps=num_steps,
                             optimizer=optimizer, criterion=loss_fn, time_var=False,
                             device=device)
    print(f"Epoch {epoch}, Train Loss: {avg_loss.item():.2f}")

Let's see the accuracy on the full test set, again using \texttt{SF.accuracy_rate}.

```python
def test_accuracy(data_loader, net, num_steps):
    with torch.no_grad():
        total = 0
        acc = 0
        net.eval()
        data_loader = iter(data_loader)
        for data, targets in data_loader:
            data = data.to(device)
            targets = targets.to(device)
            spk_rec = forward_pass(net, data, num_steps)
            acc += SF.accuracy_rate(spk_rec, targets) * spk_rec.size(1)
            total += spk_rec.size(1)
        return acc/total

print(f"Test set accuracy: {test_accuracy(test_loader, net, num_steps)*100:.3f}%")
```

More control over your model

If you are simulating more complex architectures, such as residual nets, then your best bet is to wrap the network up in a class as shown below. This time, we will explicitly use the membrane potential, \texttt{mem}, and let \texttt{init_hidden} default to false.

For the sake of speed, we’ll just simulate a fully-connected SNN, but this can be generalized to other network types (e.g., Convs).

In addition, let’s set the neuron decay rate, \texttt{beta}, to be a learnable parameter. The first layer will have a shared decay rate across neurons. Each neuron in the second layer will have an independent decay rate. The decay is clipped between \([0,1]\).
import torch.nn.functional as F

# Define Network
class Net(nn.Module):
    def __init__(self):
        super().__init__()

        num_inputs = 784
        num_hidden = 300
        num_outputs = 10
        spike_grad = surrogate.fast_sigmoid()

        # global decay rate for all leaky neurons in layer 1
        beta1 = 0.9
        # independent decay rate for each leaky neuron in layer 2: [0, 1)
        beta2 = torch.rand((num_outputs), dtype = torch.float) .to(device)

        # Init layers
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.Leaky(beta=beta1, spike_grad=spike_grad, learn_beta=True)
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        self.lif2 = snn.Leaky(beta=beta2, spike_grad=spike_grad,learn_beta=True)

    def forward(self, x):

        # reset hidden states and outputs at t=0
        mem1 = self.lif1.init_leaky()
        mem2 = self.lif2.init_leaky()

        # Record the final layer
        spk2_rec = []
        mem2_rec = []

        for step in range(num_steps):
            cur1 = self.fc1(x.flatten(1))
            spk1, mem1 = self.lif1(cur1, mem1)
            cur2 = self.fc2(spk1)
            spk2, mem2 = self.lif2(cur2, mem2)

            spk2_rec.append(spk2)
            mem2_rec.append(mem2)

        return torch.stack(spk2_rec), torch.stack(mem2_rec)

# Load the network onto CUDA if available
net = Net().to(device)

optimizer = torch.optim.Adam(net.parameters(), lr=2e-3, betas=(0.9, 0.999))
loss_fn = SF.mse_count_loss(correct_rate=0.8, incorrect_rate=0.2)

num_epochs = 1
num_steps = 100  # run for 25 time steps
loss_hist = []
acc_hist = []

# training loop
for epoch in range(num_epochs):
    for i, (data, targets) in enumerate(iter(train_loader)):
        data = data.to(device)
        targets = targets.to(device)

        net.train()
        spk_rec, _ = net(data)
        loss_val = loss_fn(spk_rec, targets)

        # Gradient calculation + weight update
        optimizer.zero_grad()
        loss_val.backward()
        optimizer.step()

        # Store loss history for future plotting
        loss_hist.append(loss_val.item())

        # print every 25 iterations
        if i % 25 == 0:
            net.eval()
            print(f"Epoch {epoch}, Iteration {i} \nTrain Loss: {loss_val.item():.2f}"")

            # check accuracy on a single batch
            acc = SF.accuracy_rate(spk_rec, targets)
            acc_hist.append(acc)
            print(f"Accuracy: {acc * 100:.2f}%\n")

        # uncomment for faster termination
        # if i == 150:
        #     break

print(f"Trained decay rate of the first layer: {net.lif1.beta:.3f}\n")
print(f"Trained decay rates of the second layer: {net.lif2.beta}\n")

def test_accuracy(data_loader, net, num_steps):
    with torch.no_grad():
        total = 0
        acc = 0
        net.eval()

        data_loader = iter(data_loader)
        for data, targets in data_loader:
            data = data.to(device)
            targets = targets.to(device)
            spk_rec, _ = net(data)
acc += SF.accuracy_rate(spk_rec, targets) * spk_rec.size(1)
total += spk_rec.size(1)

return acc/total

print(f"Test set accuracy: {test_accuracy(test_loader, net, num_steps)*100:.3f}%")

Conclusion

That’s it for the quick intro to snnTorch!

• For a detailed tutorial of spiking neurons, neural nets, encoding, and training using neuromorphic datasets, check out the snnTorch tutorial series.

• For more information on the features of snnTorch, check out the documentation at this link.

• If you have ideas, suggestions or would like to find ways to get involved, then check out the snnTorch GitHub project here.

1.11.12 Examples

Samples of code snippets demonstrating usage of various modules and functions of snnTorch can be found here. More detail is available in the tutorials.

Spiking Neurons and Networks

Examples of how to use the various spiking neuron classes to build up computational graphs in conjunction with PyTorch’s torch.nn module are provided here.

Building Networks: Lapicque’s Neuron

Building a fully-connected network using Lapicque’s neuron model.

Example:

```python
import torch
import torch.nn as nn
import snntorch as snn

beta = 0.5
R = 1
C = 1.44

batch_size = 128
num_inputs = 784
num_hidden = 1000
```
```python
num_outputs = 10

num_steps = 100

# Define Network
class Net(nn.Module):
    def __init__(self):
        super().__init__()

        # initialize layers
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.Lapique(beta=beta)
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        self.lif2 = snn.Lapique(R=R, C=C)  # lif1 and lif2 are approximately equivalent

    def forward(self, x, mem1, spk1, mem2):
        for step in range(num_steps):
            cur1 = self.fc1(x)
            spk1, mem1 = self.lif1(cur1, mem1)
            cur2 = self.fc2(spk1)
            spk2, mem2 = self.lif2(cur2, mem2)

            spk2_rec.append(spk2)
            mem2_rec.append(mem2)

        return torch.stack(spk2_rec, dim=0), torch.stack(mem2_rec, dim=0)

net = Net().to(device)
output, mem_rec = net(data.view(batch_size, -1))
```

### Building Networks: 0th Order Spike Response Model

Building a fully-connected network using the 0th Order Spike Response Model.

Example:

```python
import torch
import torch.nn as nn
import snntorch as snn

alpha = 0.9
beta = 0.8

batch_size = 128
num_inputs = 784
num_hidden = 1000
num_outputs = 10
num_steps = 100
```
# Define Network

class Net(nn.Module):
    def __init__(self):
        super().__init__()

        # initialize layers
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.Alpha(alpha=alpha, beta=beta)
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        self.lif2 = snn.Alpha(alpha=alpha, beta=beta)

    def forward(self, x):
        for step in range(num_steps):
            cur1 = self.fc1(x)
            spk1, presyn1, postsyn1, mem1 = self.lif1(cur1, presyn1, postsyn1, mem1)
            cur2 = self.fc2(spk1)
            spk2, presyn2, postsyn2, mem2 = self.lif2(cur2, presyn2, postsyn2, mem2)

            spk2_rec.append(spk2)
            mem2_rec.append(mem2)

        return torch.stack(spk2_rec, dim=0), torch.stack(mem2_rec, dim=0)

net = Net().to(device)
output, mem_rec = net(data.view(batch_size, -1))

Building Networks: Synaptic Conductance-based LIF Neuron

Building a fully-connected network using a 2nd Order Synaptic Conductance-based neuron model.

Example:

```python
import torch
import torch.nn as nn
import snntorch as snn

alpha = 0.9
beta = 0.85

batch_size = 128
num_inputs = 784
num_hidden = 1000
num_outputs = 10
num_steps = 100

# Define Network
class Net(nn.Module):
    def __init__(self):
        super().__init__()

        # initialize layers
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.Alpha(alpha=alpha, beta=beta)
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        self.lif2 = snn.Alpha(alpha=alpha, beta=beta)

    def forward(self, x):
        for step in range(num_steps):
            cur1 = self.fc1(x)
            spk1, presyn1, postsyn1, mem1 = self.lif1(cur1, presyn1, postsyn1, mem1)
            cur2 = self.fc2(spk1)
            spk2, presyn2, postsyn2, mem2 = self.lif2(cur2, presyn2, postsyn2, mem2)

            spk2_rec.append(spk2)
            mem2_rec.append(mem2)

        return torch.stack(spk2_rec, dim=0), torch.stack(mem2_rec, dim=0)

net = Net().to(device)
output, mem_rec = net(data.view(batch_size, -1))
```
super().__init__()

# initialize layers
self.fc1 = nn.Linear(num_inputs, num_hidden)
self.lif1 = snn.Synaptic(alpha=alpha, beta=beta)
self.fc2 = nn.Linear(num_hidden, num_outputs)
self.lif2 = snn.Synaptic(alpha=alpha, beta=beta)

def forward(self, x):
    spk1, syn1, mem1 = self.lif1.init_synaptic(batch_size, num_hidden)
    spk2, syn2, mem2 = self.lif2.init_synaptic(batch_size, num_outputs)
    spk2_rec = []  # Record the output trace of spikes
    mem2_rec = []  # Record the output trace of membrane potential
    for step in range(num_steps):
        cur1 = self.fc1(x)
        spk1, syn1, mem1 = self.lif1(cur1, syn1, mem1)
        cur2 = self.fc2(spk1)
        spk2, syn2, mem2 = self.lif2(cur2, syn2, mem2)
        spk2_rec.append(spk2)
        mem2_rec.append(mem2)

    return torch.stack(spk2_rec, dim=0), torch.stack(mem2_rec, dim=0)

net = Net().to(device)
output, mem_rec = net(data.view(batch_size, -1))

Building Networks with Instance Variables: Synaptic Conductance-based LIF Neuron

Building a fully-connected network using a Synaptic Conductance-based neuron model. Using instance variables are only required when calling the built-in backprop methods in snntorch.backprop.

Example:

```python
import torch
import torch.nn as nn
import snntorch as snn

alpha = 0.9
beta = 0.85

batch_size = 128
num_inputs = 784
num_hidden = 1000
num_outputs = 10
num_steps = 100
```
# Define Network

class Net(nn.Module):
    def __init__(self):
        super().__init__()

        # initialize layers
        snn.LIF.clear_instances()  # boilerplate
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.Synaptic(alpha=alpha, beta=beta, num_inputs=num_hidden, batch_size=batch_size, init_hidden=True)
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        self.lif2 = snn.Synaptic(alpha=alpha, beta=beta, num_inputs=num_outputs, batch_size=batch_size, init_hidden=True)

    # move the time-loop into the training-loop
    def forward(self, x):
        cur1 = self.fc1(x)
        self.lif1.spk1, self.lif1.syn1, self.lif1.mem1 = self.lif1(cur1, self.lif1.syn, self.lif1.mem)
        cur2 = self.fc2(self.lif1.spk)
        self.lif2.spk, self.lif2.syn, self.lif2.mem = self.lif2(cur2, self.lif2.syn, self.lif2.mem)

        return self.lif2.spk, self.lif2.mem

net = Net().to(device)

for step in range(num_steps):
    spk_out, mem_out = net(data.view(batch_size, -1))

---

## Spike Plot and Animation

Examples of how to plot and animate the responses of individual and networks of spiking neurons are provided here.

### Animator

Generate an animation by looping through the first dimension of a sample of spiking data. Time must be the first dimension of data below.

Example:

```python
import snntorch.spikeplot as splt
import matplotlib.pyplot as plt

# spike_data contains 128 samples, each of 100 time steps in duration
print(spike_data.size())
>>> torch.Size([100, 128, 1, 28, 28])
```
# Index into a single sample from a minibatch

```python
spike_data_sample = spike_data[:, 0, 0]
print(spike_data_sample.size())
```

```
>>> torch.Size([100, 28, 28])
```

# Plot

```python
fig, ax = plt.subplots()
anim = splt.animator(spike_data_sample, fig, ax)
HTML(anim.to_html5_video())
```

# Save as a gif

```python
anim.save("spike_mnist.gif")
```

## Raster

Example:

```python
import snntorch.spikeplot as splt
import matplotlib.pyplot as plt

# spike_data contains 128 samples, each of 100 time steps in duration
print(spike_data.size())
```

```
>>> torch.Size([100, 128, 1, 28, 28])
```

# Index into a single sample from a minibatch

```python
spike_data_sample = spike_data[:, 0, 0]
print(spike_data_sample.size())
```

```
>>> torch.Size([100, 28, 28])
```

```python
fig = plt.figure(facecolor="w", figsize=(10, 5))
ax = fig.add_subplot(111)

# s: size of scatter points; c: color of scatter points
splt.raster(spike_data_sample, ax, s=1.5, c="black")
plt.title("Input Layer")
plt.xlabel("Time step")
plt.ylabel("Neuron Number")
plt.show()
```

## Spike Count

Generate horizontal bar plot for a single forward pass. Options to animate are also available.

Example:

```python
import snntorch.spikeplot as splt
import matplotlib.pyplot as plt
from IPython.display import HTML
```
num_steps = 25

# Use splt.spike_count to display behavior of output neurons for a single sample during feedforward

# spk_rec is a recording of output spikes across 25 time steps, using `batch_size=128`
print(spk_rec.size())
>>> torch.Size([25, 128, 10])

# We only need a single data sample
spk_results = torch.stack(spk_rec, dim=0)[:1, 0, :].to('cpu')
print(spk_results.size())
>>> torch.Size([25, 10])

fig, ax = plt.subplots(facecolor='w', figsize=(12, 7))
labels=['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

# Plot and save spike count histogram
splt.spike_count(spk_results, fig, ax, labels, num_steps = num_steps, time_step=1e-3)
plt.show()
plt.savefig('hist2.png', dpi=300, bbox_inches='tight')

# Animate and save spike count histogram
anim = splt.spike_count(spk_results, fig, ax, labels, animate=True, interpolate=5, num_steps = num_steps, time_step=1e-3)
HTML(anim.to_html5_video())
anim.save("spike_bar.gif")

Traces

Plot an array of neuron traces (e.g., membrane potential or synaptic current). Example:

```python
import snntorch.spikeplot as splt

# mem_rec contains the traces of 9 neuron membrane potentials across 100 time steps in duration
print(mem_rec.size())
>>> torch.Size([100, 9])

# Plot
splt.traces(mem_rec, dim=(3,3))
```
Spikevision Datasets

Examples of how to use the various neuromorphic dataloaders provided in the spikevision.spikedata module.

Spikevision Datasets: DVS Gesture

DVS Gesture DataLoader

Example:

```python
import snntorch as snn
from snntorch.spikevision import spikedata
from torch.utils.data import DataLoader

# create datasets
train_ds = spikedata.DVSGesture("dataset/dvsgesture", train=True)
test_ds = spikedata.DVSGesture("dataset/dvs_gesture", train=False)

# create dataloaders
train_dl = DataLoader(train_ds, shuffle=True, batch_size=64)
test_dl = DataLoader(test_ds, shuffle=False, batch_size=64)
```

Visualizing the data:

```python
import matplotlib.pyplot as plt
import snntorch.spikeplot as splt
from IPython.display import HTML

# choose a random sample
n = 125

# index into a single sample and sum the on/off channels
a = (train_dl.dataset[n][0][:, 0] + train_dl.dataset[n][0][:, 1])

# Plot
fig, ax = plt.subplots()
anim = splt.animator(a, fig, ax, interval=10)
HTML(anim.to_html5_video())
anim.save('dvsgesture_animator.mp4', writer='ffmpeg', fps=50)
```

Spikevision Datasets: NMNIST

Neuromorphic MNIST DataLoader

Example:

```python
import snntorch as snn
from snntorch.spikevision import spikedata
from torch.utils.data import DataLoader
```

(continues on next page)
# create datasets
train_ds = spikedata.NMNIST("dataset/nmnist", train=True)
test_ds = spikedata.NMNIST("dataset/nmnist", train=False)

# create dataloaders
train_dl = DataLoader(train_ds, shuffle=True, batch_size=64)
test_dl = DataLoader(test_ds, shuffle=False, batch_size=64)

Visualizing the data:

```python
import matplotlib.pyplot as plt
import snntorch.spikeplot as splt
from IPython.display import HTML

# choose a random sample
n = 40000

# index into a single sample and sum the on/off channels
a = (train_dl.dataset[n][0][0] + train_dl.dataset[n][0][1])

# Plot
fig, ax = plt.subplots()
anim = splt.animator(a, fig, ax, interval=10)
HTML(anim.to_html5_video())
anim.save('nmnist_animator.mp4', writer='ffmpeg', fps=50)
```

**Spikevision Datasets: SHD Dataset**

Spiking Heidelberg Digits Dataset DataLoader

Example:

```python
import snntorch as snn
from snntorch.spikevision import spikedata

from torch.utils.data import DataLoader

# create datasets
train_ds = spikedata.SHD("dataset/shd", train=True)
test_ds = spikedata.SHD("dataset/shd", train=False)

# create dataloaders
train_dl = DataLoader(train_ds, shuffle=True, batch_size=64)
test_dl = DataLoader(test_ds, shuffle=False, batch_size=64)

Visualizing the data:

```python
import matplotlib.pyplot as plt
import snntorch.spikeplot as splt
```
# choose a random sample
n = 6295

# initialize figure and axes
fig = plt.figure(facecolor="w", figsize=(10, 5))
ax = fig.add_subplot(111)

# use spikeplot to generate a raster
splt.raster(train_dl.dataset[n][0], ax, s=1.5, c="black")

## Surrogate Gradients

Examples of how to override the default backward function of a spiking neuron are provided here.

### Fast Sigmoid

There are two ways to apply the Fast Sigmoid surrogate gradient:

```python
import torch.nn as nn
import snntorch as snn
from snntorch import surrogate

alpha = 0.6
beta = 0.5
num_inputs = 784
num_hidden = 1000
num_outputs = 10

Example:

# Method 1 uses a closure to wrap around FastSigmoid, bundling it with the specified
# slope before calling it

# initialize layers
fc1 = nn.Linear(num_inputs, num_hidden)
lif1 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.fast_sigmoid(slope=50))
fc2 = nn.Linear(num_hidden, num_outputs)
lif2 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.fast_sigmoid(slope=50))

Example:

# Method 2 applies the autograd inherited method directly, using the default value of
# slope=25
# The default value could also be called by specifying `fast_sigmoid()` instead

# initialize layers
fc1 = nn.Linear(num_inputs, num_hidden)
lif1 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.FastSigmoid.apply)
fc2 = nn.Linear(num_hidden, num_outputs)
lif2 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.FastSigmoid.apply)
```
Leaky Spike Operator

There are two ways to apply the Leaky Spike Operator surrogate gradient:

```python
import torch.nn as nn
import snntorch as snn
from snntorch import surrogate

alpha = 0.6
beta = 0.5

num_inputs = 784
num_hidden = 1000
num_outputs = 10
```

Example:

```python
# Method 1 uses a closure to wrap around LeakySpikeOperator, bundling it with the specified slope before calling it

# initialize layers
fc1 = nn.Linear(num_inputs, num_hidden)
lif1 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.LSO(slope=0.2))
fc2 = nn.Linear(num_hidden, num_outputs)
lif2 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.LSO(slope=0.2))
```

Example:

```python
# Method 2 applies the autograd inherited method directly, using the default values of slope=0.2
# The default value could also be called by specifying `LSO()` instead

# initialize layers
fc1 = nn.Linear(num_inputs, num_hidden)
lif1 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.LeakySpikeOperator.apply)
fc2 = nn.Linear(num_hidden, num_outputs)
lif2 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.LeakySpikeOperator.apply)
```

Sigmoid

There are two ways to apply the Sigmoid surrogate gradient:

```python
import torch.nn as nn
import snntorch as snn
from snntorch import surrogate

alpha = 0.6
beta = 0.5

num_inputs = 784
```

(continues on next page)
num_hidden = 1000  
num_outputs = 10

Example:

```python
# Method 1 uses a closure to wrap around Sigmoid, bundling it with the specified slope, before calling it

# initialize layers
fc1 = nn.Linear(num_inputs, num_hidden)  
lif1 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.sigmoid(slope=50))  
f2 = nn.Linear(num_hidden, num_outputs)  
lif2 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.sigmoid(slope=50))
```

Example:

```python
# Method 2 applies the autograd inherited method directly, using the default value of slope=25
# The default value could also be called by specifying `sigmoid()` instead

# initialize layers
fc1 = nn.Linear(num_inputs, num_hidden)  
lif1 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.Sigmoid.apply)  
f2 = nn.Linear(num_hidden, num_outputs)  
lif2 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.Sigmoid.apply)
```

**Spike Rate Escape**

There are two ways to apply the Spike Rate Escape surrogate gradient:

```python
import torch.nn as nn
import snntorch as snn
from snntorch import surrogate

alpha = 0.6  
beta = 0.5  

num_inputs = 784  
nnum_hidden = 1000  
nnum_outputs = 10

Example:

```python
# Method 1 uses a closure to wrap around SpikeRateEscape, bundling it with the specified beta and slope before calling it

# initialize layers
fc1 = nn.Linear(num_inputs, num_hidden)  
lif1 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.spike_rate_escape(beta=2, slope=50))  
f2 = nn.Linear(num_hidden, num_outputs)  
lif2 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.spike_rate_escape(beta=2, slope=50))
```

(continues on next page)
lif2 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.spike_rate_escape(beta=2, slope=25))

Example:

```python
# Method 2 applies the autograd inherited method directly, using the default values of beta=1 and slope=25
# The default value could also be called by specifying `spike_rate_escape()` instead
# initialize layers
fc1 = nn.Linear(num_inputs, num_hidden)
lif1 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.SpikeRateEscape.apply)
fc2 = nn.Linear(num_hidden, num_outputs)
lif2 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.SpikeRateEscape.apply)
```

**Stochastic Spike Operator**

There are two ways to apply the Stochastic Spike Operator surrogate gradient:

```python
import torch.nn as nn
import snntorch as snn
from snntorch import surrogate

alpha = 0.6
beta = 0.5
num_inputs = 784
num_hidden = 1000
num_outputs = 10

Example:

# Method 1 uses a closure to wrap around StochasticSpikeOperator, bundling it with the specified mean and variance before calling it
# initialize layers
fc1 = nn.Linear(num_inputs, num_hidden)
lif1 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.SSO(mean=0.1, variance=0.1))
fc2 = nn.Linear(num_hidden, num_outputs)
lif2 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.SSO(mean=0.1, variance=0.1))
```

Example:

```python
# Method 2 applies the autograd inherited method directly, using the default values of mean=0, variance=0.2
# The default value could also be called by specifying `SSO()` instead
# initialize layers
fc1 = nn.Linear(num_inputs, num_hidden)
```
lif1 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.StochasticSpikeOperator._apply)
fc2 = nn.Linear(num_hidden, num_outputs)
lif2 = snn.Synaptic(alpha=alpha, beta=beta, spike_grad=surrogate.StochasticSpikeOperator._apply)

1.11.13 Tutorials

The tutorial consists of a series of Google Colab notebooks. Static non-editable versions are also available.

<table>
<thead>
<tr>
<th>Tutorial</th>
<th>Title</th>
<th>Colab Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutorial 1</td>
<td>Spike Encoding with snnTorch</td>
<td></td>
</tr>
<tr>
<td>Tutorial 2</td>
<td>The Leaky Integrate and Fire Neuron</td>
<td></td>
</tr>
<tr>
<td>Tutorial 3</td>
<td>A Feedforward Spiking Neural Network</td>
<td></td>
</tr>
<tr>
<td>Tutorial 4</td>
<td>2nd Order Spiking Neuron Models (Optional)</td>
<td></td>
</tr>
<tr>
<td>Tutorial 5</td>
<td>Training Spiking Neural Networks with snnTorch</td>
<td></td>
</tr>
<tr>
<td>Tutorial 6</td>
<td>Surrogate Gradient Descent in a Convolutional SNN</td>
<td></td>
</tr>
<tr>
<td>Tutorial 7</td>
<td>Neuromorphic Datasets with Tonic + snnTorch</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Advanced Tutorials</th>
<th>Colab Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Coding</td>
<td></td>
</tr>
<tr>
<td>Accelerating snnTorch on IPUs</td>
<td>—</td>
</tr>
</tbody>
</table>

Future tutorials on spiking neurons and training are under construction.

**Tutorial 1 - Spike Encoding**

Tutorial written by Jason K. Eshraghian (www.ncg.ucsc.edu)

The snnTorch tutorial series is based on the following paper. If you find these resources or code useful in your work, please consider citing the following source:


**Note:**

This tutorial is a static non-editable version. Interactive, editable versions are available via the following links:

- Google Colab
- Local Notebook (download via GitHub)

In this tutorial, you will learn how to use snnTorch to:

- convert datasets into spiking datasets,
- how to visualise them,
and how to generate random spike trains.

Introduction

Light is what we see when the retina converts photons into spikes. Odors are what we smell when volatilised molecules are converted into spikes. Touch is what we feel when nerve endings turn tactile pressure into spikes. The brain trades in the global currency of the *spike*.

If our end goal is to build a spiking neural network (SNN), it makes sense to use spikes at the input too. Although it is quite common to use non-spiking inputs (as will be seen in Tutorial 3), part of the appeal of encoding data come from the *three S’s*: spikes, sparsity, and static suppression.

1. **Spikes**: (a-b) Biological neurons process and communicate via spikes, which are electrical impulses of approximately 100 mV in amplitude. (c) Many computational models of neurons simplify this voltage burst to a discrete, single-bit event: a ‘1’ or a ‘0’. This is far simpler to represent in hardware than a high precision value.

2. **Sparsity**: (c) Neurons spend most of their time at rest, silencing most activations to zero at any given time. Not only are sparse vectors/tensors (with loads of zeros) cheap to store, but say we need to multiply sparse activations with synaptic weights. If most values are multiplied by ‘0’, then we don’t need to read many of the network parameters from memory. This means neuromorphic hardware can be extremely efficient.

3. **Static-Suppression (a.k.a, event-driven processing)**: (d-e) The sensory periphery only processes information when there is new information to process. Each pixel in (e) responds to changes in illuminance, so most of the image is blocked out. Conventional signal processing requires all channels/pixels to adhere to a global sampling/shutter rate, which slows down how frequently sensing can take place. Event-driven processing now only contributes to sparsity and power-efficiency by blocking unchanging input, but it often allows for much faster processing speeds.
In this tutorial, we will assume we have some non-spiking input data (i.e., the MNIST dataset) and that we want to encode it into spikes using a few different techniques. So let’s get started!

Install the latest PyPi distribution of snnTorch:

```bash
$ pip install snntorch
```
1. Setting up the MNIST Dataset

1.1. Import packages and setup environment

```python
import snntorch as snn
import torch

# Training Parameters
batch_size=128
data_path='./data/mnist'
um_classes = 10 # MNIST has 10 output classes

# Torch Variables
dtype = torch.float
```

1.2 Download Dataset

```python
from torchvision import datasets, transforms

# Define a transform
transform = transforms.Compose([transforms.Resize((28,28)),
                                 transforms.Grayscale(),
                                 transforms.ToTensor(),
                                 transforms.Normalize((0,), (1,))])

mnist_train = datasets.MNIST(data_path, train=True, download=True, transform=transform)

If the above code block throws an error, e.g. the MNIST servers are down, then uncomment the following code instead.

```bash
# # temporary dataloader if MNIST service is unavailable
# !wget www.di.ens.fr/~lelarge/MNIST.tar.gz
# !tar -zxvf MNIST.tar.gz

# mnist_train = datasets.MNIST(root = './', train=True, download=True, transform=transform)
```

Until we actually start training a network, we won't need large datasets. `snntorch.utils` contains a few useful functions for modifying datasets. Apply `data_subset` to reduce the dataset by the factor defined in `subset`. E.g., for `subset=10`, a training set of 60,000 will be reduced to 6,000.

```python
from snntorch import utils

subset = 10
mnist_train = utils.data_subset(mnist_train, subset)

>>> print(f"The size of mnist_train is {len(mnist_train)}")
The size of mnist_train is 6000
1.3 Create DataLoader

The Dataset objects created above load data into memory, and the DataLoader will serve it up in batches. DataLoaders in PyTorch are a handy interface for passing data into a network. They return an iterator divided up into mini-batches of size \texttt{batch_size}.

```python
from torch.utils.data import DataLoader

train_loader = DataLoader(mnist_train, batch_size=batch_size, shuffle=True)
```

2. Spike Encoding

Spiking Neural Networks (SNNs) are made to exploit time-varying data. And yet, MNIST is not a time-varying dataset. There are two options for using MNIST with an SNN:

1. Repeatedly pass the same training sample $X \in \mathbb{R}^{m \times n}$ to the network at each time step. This is like converting MNIST into a static, unchanging video. Each element of $X$ can take a high precision value normalized between 0 and 1: $X_{ij} \in [0, 1]$.

2. Convert the input into a spike train of sequence length \texttt{num_steps}, where each feature/pixel takes on a discrete value $X_{i,j} \in \{0, 1\}$. In this case, MNIST is converted into a time-varying sequence of spikes that features a relation to the original image.
The first method is quite straightforward, and does not fully exploit the temporal dynamics of SNNs. So let’s consider data-to-spike conversion (encoding) from (2) in more detail.

The module \texttt{snntorch.spikegen} (i.e., spike generation) contains a series of functions that simplify the conversion of data into spikes. There are currently three options available for spike encoding in \texttt{snntorch}:

1. Rate coding: \texttt{spikegen.rate}
2. Latency coding: \texttt{spikegen.latency}
3. Delta modulation: \texttt{spikegen.delta}

How do these differ?

1. \textit{Rate coding} uses input features to determine spiking \textbf{frequency}
2. \textit{Latency coding} uses input features to determine spike \textbf{timing}
3. \textit{Delta modulation} uses the temporal \textbf{change} of input features to generate spikes

\subsection*{2.1 Rate coding of MNIST}

One example of converting input data into a rate code is as follows. Each normalised input feature $X_{ij}$ is used as the probability an event (spike) occurs at any given time step, returning a rate-coded value $R_{ij}$. This can be treated as a Bernoulli trial: $R_{ij} \sim B(n, p)$, where the number of trials is $n = 1$, and the probability of success (spiking) is $p = X_{ij}$. Explicitly, the probability a spike occurs is:

$$P(R_{ij} = 1) = X_{ij} = 1 - P(R_{ij} = 0)$$

Create a vector filled with the value ‘0.5’ and encode it using the above technique:
raw_vector = torch.ones(num_steps)*0.5

# pass each sample through a Bernoulli trial
rate_coded_vector = torch.bernoulli(raw_vector)

::

>>> print(f"Converted vector: {rate_coded_vector}")
Converted vector: tensor([1., 1., 1., 0., 0., 1., 1., 0., 1., 0.])

>>> print(f"The output is spiking {rate_coded_vector.sum()*100/len(rate_coded_vector):.2f}% of the time.")
The output is spiking 60.00% of the time.

Now try again, but increasing the length of raw_vector:

num_steps = 100

# create vector filled with 0.5
raw_vector = torch.ones(num_steps)*0.5

# pass each sample through a Bernoulli trial
rate_coded_vector = torch.bernoulli(raw_vector)

>>> print(f"The output is spiking {rate_coded_vector.sum()*100/len(rate_coded_vector):.2f}% of the time.")
The output is spiking 48.00% of the time.

As num_steps→∞, the proportion of spikes approaches the original raw value.

For an MNIST image, this probability of spiking corresponds to the pixel value. A white pixel corresponds to a 100% probability of spiking, and a black pixel will never generate a spike. Take a look at the ‘Rate Coding’ column below for further intuition.
In a similar way, `spikegen.rate` can be used to generate a rate-coded sample of data. As each sample of MNIST is just an image, we can use `num_steps` to repeat it across time.

```python
from snntorch import spikegen

# Iterate through minibatches
data = iter(train_loader)
data_it, targets_it = next(data)

# Spiking Data
spike_data = spikegen.rate(data_it, num_steps=num_steps)
```

If the input falls outside of [0, 1], this no longer represents a probability. Such cases are automatically clipped to ensure the feature represents a probability.

The structure of the input data is `[num_steps x batch_size x input dimensions]`:

```python
>>> print(spike_data.size())
torch.Size([100, 128, 1, 28, 28])
```

2.2 Visualization

2.2.1 Animation

`snnTorch` contains a module `snntorch.spikeplot` that simplifies the process of visualizing, plotting, and animating spiking neurons.

```python
import matplotlib.pyplot as plt
import snntorch.spikeplot as splt
from IPython.display import HTML

To plot one sample of data, index into a single sample from the batch (B) dimension of `spike_data`, `[T x B x 1 x 28 x 28]`:

```python
spike_data_sample = spike_data[:, 0, 0]
>>> print(spike_data_sample.size())
torch.Size([100, 28, 28])
```

`spikeplot.animator` makes it super simple to animate 2-D data. Note: if you are running the notebook locally on your desktop, please uncomment the line below and modify the path to your ffmpeg.exe

```python
fig, ax = plt.subplots()
anim = splt.animator(spike_data_sample, fig, ax)
# plt.rcParams['animation.ffmpeg_path'] = 'C:\path\to\your\ffmpeg.exe'

HTML(anim.to_html5_video())
```

# If you're feeling sentimental, you can save the animation: .gif, .mp4 etc.
anim.save("spike_mnist_test.mp4")

The associated target label can be indexed as follows:
The corresponding target is: 7

MNIST features a greyscale image, and the white text guarantees a 100% of spiking at every time step. So let's do that again but reduce the spiking frequency. This can be achieved by setting the argument gain. Here, we will reduce spiking frequency to 25%.

```python
spike_data = spikegen.rate(data_it, num_steps=num_steps, gain=0.25)
spike_data_sample2 = spike_data[:, 0, 0]
fig, ax = plt.subplots()
anim = splt.animator(spike_data_sample2, fig, ax)
HTML(anim.to_html5_video())
```

# Uncomment for optional save
# anim.save("spike_mnist_test2.mp4")

Now average the spikes out over time and reconstruct the input images.

```python
plt.figure(facecolor="w")
plt.subplot(1,2,1)
plt.imshow(spike_data_sample.mean(axis=0).reshape((28,-1)).cpu(), cmap='binary')
plt.axis('off')
plt.title('Gain = 1')

plt.subplot(1,2,2)
plt.imshow(spike_data_sample2.mean(axis=0).reshape((28,-1)).cpu(), cmap='binary')
plt.axis('off')
plt.title('Gain = 0.25')

plt.show()
```

The case where gain=0.25 is lighter than where gain=1, as spiking probability has been reduced by a factor of $\times 4$. 

Gain = 1  
Gain = 0.25

```

\[ \text{Gain = 1} \quad \text{Gain = 0.25} \]

\[
\begin{array}{c}
\text{3} \\
\text{3}
\end{array}
\]
2.2.2 Raster Plots

Alternatively, we can generate a raster plot of an input sample. This requires reshaping the sample into a 2-D tensor, where 'time' is the first dimension. Pass this sample into the function `spikeplot.raster`.

```python
# Reshape
spike_data_sample2 = spike_data_sample2.reshape((num_steps, -1))

# raster plot
fig = plt.figure(facecolor="w", figsize=(10, 5))
ax = fig.add_subplot(111)
splt.raster(spike_data_sample2, ax, s=1.5, c="black")
plt.title("Input Layer")
plt.xlabel("Time step")
plt.ylabel("Neuron Number")
plt.show()
```

The following code snippet shows how to index into one single neuron. Depending on the input data, you may need to try a few different neurons between 0 & 784 before finding one that spikes.

```python
::
idx = 210 # index into 210th neuron
fig = plt.figure(facecolor="w", figsize=(8, 1))
ax = fig.add_subplot(111)
splt.raster(spike_data_sample.reshape(num_steps, -1)[:, idx].unsqueeze(1), ax, s=100, c="black", marker="|")
plt.title("Input Neuron")
plt.xlabel("Time step")
plt.yticks([])
plt.show()
```
2.2.3 Summary of Rate Coding

The idea of rate coding is actually quite controversial. Although we are fairly confident rate coding takes place at our sensory periphery, we are not convinced that the cortex globally encodes information as spike rates. A couple of compelling reasons why include:

- **Power Consumption**: Nature optimised for efficiency. Multiple spikes are needed to achieve any sort of task, and each spike consumes power. In fact, Olshausen and Field’s work in “What is the other 85% of V1 doing?” demonstrates that rate-coding can only explain, at most, the activity of 15% of neurons in the primary visual cortex (V1). It is unlikely to be the only mechanism within the brain, which is both resource-constrained and highly efficient.

- **Reaction Response Times**: We know that the reaction time of a human is roughly around 250ms. If the average firing rate of a neuron in the human brain is on the order of 10Hz, then we can only process about 2 spikes within our reaction timescale.

So why, then, might we use rate codes if they are not optimal for power efficiency or latency? Even if our brain doesn’t process data as a rate, we are fairly sure that our biological sensors do. The power/latency disadvantages are partially offset by showing huge noise robustness: it’s fine if some of the spikes fail to generate, because there will be plenty more where they came from.

Additionally, you may have heard of the Hebbian mantra of “neurons that fire together, wire together”. If there is plenty of spiking, this may suggest there is plenty of learning. In some cases where training SNNs proves to be challenging, encouraging more firing via a rate code is one possible solution.

Rate coding is almost certainly working in conjunction with other encoding schemes in the brain. We will consider these other encoding mechanisms in the following sections. This covers the `spikegen.rate` function. Further information can be found in the documentation here.

2.3 Latency Coding of MNIST

Temporal codes capture information about the precise firing time of neurons; a single spike carries much more meaning than in rate codes which rely on firing frequency. While this opens up more susceptibility to noise, it can also decrease the power consumed by the hardware running SNN algorithms by orders of magnitude.

`spikegen.latency` is a function that allows each input to fire at most once during the full time sweep. Features closer to 1 will fire earlier and features closer to 0 will fire later. I.e., in our MNIST case, bright pixels will fire earlier and dark pixels will fire later.

The following block derives how this works. If you’ve forgotten circuit theory and/or the math means nothing to you, then don’t worry! All that matters is: **big** input means **fast** spike; **small** input means **late** spike.
Optional: Derivation of Latency Code Mechanism

By default, spike timing is calculated by treating the input feature as the current injection $I_{in}$ into an RC circuit. This current moves charge onto the capacitor, which increases $V(t)$. We assume that there is a trigger voltage, $V_{thr}$, which once reached, generates a spike. The question then becomes: for a given input current (and equivalently, input feature), how long does it take for a spike to be generated?

Starting with Kirchhoff’s current law, $I_{in} = I_R + I_C$, the rest of the derivation leads us to a logarithmic relationship between time and the input.

The following function uses the above result to convert a feature of intensity $X_{ij} \in [0, 1]$ into a latency coded response.
L_{ij}.

```python
def convert_to_time(data, tau=5, threshold=0.01):
    spike_time = tau * torch.log(data / (data - threshold))
    return spike_time
```

Now, use the above function to visualize the relationship between input feature intensity and its corresponding spike time.

```python
raw_input = torch.arange(0, 5, 0.05)  # tensor from 0 to 5
spike_times = convert_to_time(raw_input)
plt.plot(raw_input, spike_times)
plt.xlabel('Input Value')
plt.ylabel('Spike Time (s)')
plt.show()
```

The smaller the value, the later the spike occurs with exponential dependence.

The vector `spike_times` contains the time at which spikes are triggered, rather than a sparse tensor that contains the spikes themselves (1’s and 0’s). When running an SNN simulation, we need the 1/0 representation to obtain all of the advantages of using spikes. This whole process can be automated using `spikegen.latency`, where we pass a minibatch from the MNIST dataset in `data_it`:

```python
spike_data = spikegen.latency(data_it, num_steps=100, tau=5, threshold=0.01)
```

Some of the arguments include:

- **tau**: the RC time constant of the circuit. By default, the input features are treated as a constant current injected into an RC circuit. A higher `tau` will induce slower firing.
- **threshold**: the membrane potential firing threshold. Input values below this threshold do not have a closed-form solution, as the input current is insufficient to drive the membrane up to the threshold. All values below the threshold are clipped and assigned to the final time step.
2.3.1 Raster plot

```python
fig = plt.figure(facecolor="w", figsize=(10, 5))
ax = fig.add_subplot(111)
splt.raster(spike_data[:, 0].view(num_steps, -1), ax, s=25, c="black")
plt.title("Input Layer")
plt.xlabel("Time step")
plt.ylabel("Neuron Number")
plt.show()

# optional save
# fig.savefig('destination_path.png', format='png', dpi=300)
```

To make sense of the raster plot, note that high intensity features fire first, whereas low intensity features fire last:
The logarithmic code coupled with the lack of diverse input values (i.e., the lack of midtone/grayscale features) causes significant clustering in two areas of the plot. The bright pixels induce firing at the start of the run, and the dark pixels at the end. We can increase $\tau$ to slow down the spike times, or linearize the spike times by setting the optional argument linear=True.

```python
spike_data = spikegen.latency(data_it, num_steps=100, tau=5, threshold=0.01, linear=True)

fig = plt.figure(facecolor="w", figsize=(10, 5))
ax = fig.add_subplot(111)
splt.raster(spike_data[:, 0].view(num_steps, -1), ax, s=25, c="black")
plt.title("Input Layer")
plt.xlabel("Time step")
plt.ylabel("Neuron Number")
plt.show()
```
The spread of firing times is much more evenly distributed now. This is achieved by linearizing the logarithmic equation according to the rules shown below. Unlike the RC model, there is no physical basis for the model. It’s just simpler.

But note how all firing occurs within the first ~5 time steps, whereas the simulation range is 100 time steps. This indicates there are many redundant time steps doing nothing. This can be solved by either increasing \( \tau \) to slow down the time constant, or setting the optional argument `normalize=True` to span the full range of `num_steps`.

```python
spike_data = spikegen.latency(data_it, num_steps=100, tau=5, threshold=0.01, normalize=True, linear=True)
```

(continues on next page)
splt.raster(spike_data[:, 0].view(num_steps, -1), ax, s=25, c="black")
plt.title("Input Layer")
plt.xlabel("Time step")
plt.ylabel("Neuron Number")
plt.show()

One major advantage of latency coding over rate coding is sparsity. If neurons are constrained to firing a maximum of once over the time course of interest, then this promotes low-power operation.

In the scenario shown above, a majority of the spikes occur at the final time step, where the input features fall below the threshold. In a sense, the dark background of the MNIST sample holds no useful information.

We can remove these redundant features by setting clip=True.

```python
spike_data = spikegen.latency(data_it, num_steps=100, tau=5, threshold=0.01, clip=True, normalize=True, linear=True)
fig = plt.figure(facecolor="w", figsize=(10, 5))
ax = fig.add_subplot(111)
splt.raster(spike_data[:, 0].view(num_steps, -1), ax, s=25, c="black")
plt.title("Input Layer")
plt.xlabel("Time step")
plt.ylabel("Neuron Number")
plt.show()
```
2.3.2 Animation

We will run the exact same code block as before to create an animation.

```
>>> spike_data_sample = spike_data[:, 0, 0]
>>> print(spike_data_sample.size())
torch.Size([100, 28, 28])
```

```
fig, ax = plt.subplots()
anim = splt.animator(spike_data_sample, fig, ax)

HTML(anim.to_html5_video())
```

This animation is obviously much tougher to make out in video form, but a keen eye will be able to catch a glimpse of the initial frame where most of the spikes occur. Index into the corresponding target value to check its value.

```
# Save output: .gif, .mp4 etc.
# anim.save("mnist_latency.gif")
```

```
>>> print(targets_it[0])
tensor(4, device='cuda:0')
```

That’s it for the spikegen.latency function. Further information can be found in the documentation here.
2.4 Delta Modulation

There are theories that the retina is adaptive: it will only process information when there is something new to process. If there is no change in your field of view, then your photoreceptor cells are less prone to firing.

That is to say: biology is event-driven. Neurons thrive on change.

As a nifty example, a few researchers have dedicated their lives to designing retina-inspired image sensors, for example, the Dynamic Vision Sensor. Although the attached link is from over a decade ago, the work in this video was ahead of its time.

Delta modulation is based on event-driven spiking. The `snntorch.delta` function accepts a time-series tensor as input. It takes the difference between each subsequent feature across all time steps. By default, if the difference is both positive and greater than the threshold $V_{th}$, a spike is generated:
To illustrate, let’s first come up with a contrived example where we create our own input tensor.

```python
# Create a tensor with some fake time-series data
data = torch.Tensor([0, 1, 0, 2, 8, -20, 20, -5, 0, 1, 0])

# Plot the tensor
plt.plot(data)
plt.title("Some fake time-series data")
plt.xlabel("Time step")
```

(continues on next page)
plt.ylabel("Voltage (mV)")
plt.show()

Pass the above tensor into the `spikegen.delta` function, with an arbitrarily selected `threshold=4`:

```python
# Convert data
spike_data = spikegen.delta(data, threshold=4)

# Create fig, ax
fig = plt.figure(facecolor="w", figsize=(8, 1))
ax = fig.add_subplot(111)

# Raster plot of delta converted data
splt.raster(spike_data, ax, c="black")

plt.title("Input Neuron")
plt.xlabel("Time step")
plt.xticks([])
plt.xlim(0, len(data))
plt.show()
```

There are three time steps where the difference between \(data[T]\) and \(data[T+1]\) is greater than or equal to \(V_{th} = 4\). This means there are three on-spikes.

The large dip to \(-20\) has not been captured in our spikes. It might be that we care about negative swings as well, in which case we can enable the optional argument `off_spike=True`.

```python
# Convert data
spike_data = spikegen.delta(data, threshold=4, off_spike=True)

# Create fig, ax
fig = plt.figure(facecolor="w", figsize=(8, 1))
```
ax = fig.add_subplot(111)

# Raster plot of delta converted data
splt.raster(spike_data, ax, c="black")

plt.title("Input Neuron")
plt.xlabel("Time step")
plt.yticks([])
plt.xlim(0, len(data))
plt.show()

We've generated additional spikes, but this isn't actually the full picture!

Printing out the tensor will show the presence of “off-spikes” which take on a value of -1.

```python
>>> print(spike_data)
tensor([[ 0.,  0.,  0.,  0.,  1., -1.,  1., -1.,  1.,  0.,  0.])
```

While spikegen.delta has only been demonstrated on a fake sample of data, its true use is to compress time-series data by only generating spikes for sufficiently large changes/events.

That wraps up the three main spike conversion functions! There are still additional features to each of the three conversion techniques that have not been detailed in this tutorial. In particular, we have only looked at encoding input data; we have not considered how we might encode targets, and when that is necessary. We recommend referring to the documentation for a deeper dive.

### 3. Spike Generation (Optional)

Now what if we don’t actually have any data to start with? Say we just want a randomly generated spike train from scratch. Inside of spikegen.rate is a nested function, rate_conv, which actually performs the spike conversion step.

All we have to do is initialize a randomly generated torchTensor to pass in.

```python
# Create a random spike train
spike_prob = torch.rand((num_steps, 28, 28), dtype=dtype) * 0.5
spike_rand = spikegen.rate_conv(spike_rand)
```
3.1 Animation

```python
fig, ax = plt.subplots()
anim = plt.animations.spike_rand(fig, ax)

HTML(anim.to_html5_video())
```

# Save output: .gif, .mp4 etc.
# anim.save("random_spikes.gif")

3.2 Raster

```python
fig = plt.figure(facecolor="w", figsize=(10, 5))
ax = fig.add_subplot(111)
plt.raster(spike_rand[:, 0].view(num_steps, -1), ax, s=25, c="black")

plt.title("Input Layer")
plt.xlabel("Time step")
plt.ylabel("Neuron Number")
plt.show()
```
Conclusion

That’s it for spike conversion and generation. This approach generalizes beyond images, to single-dimensional and multi-dimensional tensors.

If you like this project, please consider starring the repo on GitHub as it is the easiest and best way to support it.

For reference, the documentation for spikegen can be found here and for spikeplot, here.

In the next tutorial, you will learn the basics of spiking neurons and how to use them.

Additional Resources

• Check out the snnTorch GitHub project here.

Tutorial 2 - The Leaky Integrate-and-Fire Neuron

Tutorial written by Jason K. Eshraghian (www.ncg.ucsc.edu)

The snnTorch tutorial series is based on the following paper. If you find these resources or code useful in your work, please consider citing the following source:


Note:

This tutorial is a static non-editable version. Interactive, editable versions are available via the following links:

• Google Colab
• Local Notebook (download via GitHub)

Introduction

In this tutorial, you will:

• Learn the fundamentals of the leaky integrate-and-fire (LIF) neuron model
• Use snnTorch to implement a first order LIF neuron

Install the latest PyPi distribution of snnTorch:

$ pip install snntorch

# imports
import snntorch as snn
from snntorch import spikeplot as splt
from snntorch import spikegen
import torch

(continues on next page)
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt

1. The Spectrum of Neuron Models

A large variety of neuron models are out there, ranging from biophysically accurate models (i.e., the Hodgkin-Huxley models) to the extremely simple artificial neuron that pervades all facets of modern deep learning.

**Hodgkin-Huxley Neuron Models**—While biophysical models can reproduce electrophysiological results with a high degree of accuracy, their complexity makes them difficult to use at present.

**Artificial Neuron Model**—On the other end of the spectrum is the artificial neuron. The inputs are multiplied by their corresponding weights and passed through an activation function. This simplification has enabled deep learning researchers to perform incredible feats in computer vision, natural language processing, and many other machine learning-domain tasks.

**Leaky Integrate-and-Fire Neuron Models**—Somewhere in the middle of the divide lies the leaky integrate-and-fire (LIF) neuron model. It takes the sum of weighted inputs, much like the artificial neuron. But rather than passing it directly to an activation function, it will integrate the input over time with a leakage, much like an RC circuit. If the integrated value exceeds a threshold, then the LIF neuron will emit a voltage spike. The LIF neuron abstracts away the shape and profile of the output spike; it is simply treated as a discrete event. As a result, information is not stored within the spike, but rather the timing (or frequency) of spikes. Simple spiking neuron models have produced much insight into the neural code, memory, network dynamics, and more recently, deep learning. The LIF neuron sits in the sweet spot between biological plausibility and practicality.

The different versions of the LIF model each have their own dynamics and use-cases. snnTorch currently supports the following LIF neurons:
• Lapicque’s RC model: \texttt{snntorch.Lapicque}
• 1st-order model: \texttt{snntorch.Leaky}
• Synaptic Conductance-based neuron model: \texttt{snntorch.Synaptic}
• Recurrent 1st-order model: \texttt{snntorch.RLeaky}
• Recurrent Synaptic Conductance-based neuron model: \texttt{snntorch.RSynaptic}
• Alpha neuron model: \texttt{snntorch.Alpha}

Several other non-LIF spiking neurons are also available. This tutorial focuses on the first of these models. This will be used to build towards the other models in subsequent tutorials.

\section*{2. The Leaky Integrate-and-Fire Neuron Model}

\subsection*{2.1 Spiking Neurons: Intuition}

In our brains, a neuron might be connected to 1,000 – 10,000 other neurons. If one neuron spikes, all downhill neurons might feel it. But what determines whether a neuron spikes in the first place? The past century of experiments demonstrate that if a neuron experiences \textit{sufficient} stimulus at its input, then it might become excited and fire its own spike.

Where does this stimulus come from? It could be from:

• the sensory periphery,
• an invasive electrode artificially stimulating the neuron, or in most cases,
• from other pre-synaptic neurons.
Given that these spikes are very short bursts of electrical activity, it is quite unlikely for all input spikes to arrive at the neuron body in precise unison. This indicates the presence of temporal dynamics that ‘sustain’ the input spikes, kind of like a delay.

2.2 The Passive Membrane

Like all cells, a neuron is surrounded by a thin membrane. This membrane is a lipid bilayer that insulates the conductive saline solution within the neuron from the extracellular medium. Electrically, the two conductive solutions separated by an insulator act as a capacitor.

Another function of this membrane is to control what goes in and out of this cell (e.g., ions such as Na⁺). The membrane is usually impermeable to ions which blocks them from entering and exiting the neuron body. But there are specific channels in the membrane that are triggered to open by injecting current into the neuron. This charge movement is electrically modelled by a resistor.
The following block will derive the behaviour of a LIF neuron from scratch. If you’d prefer to skip the math, then feel free to scroll on by; we’ll take a more hands-on approach to understanding the LIF neuron dynamics after the derivation.

Optional: Derivation of LIF Neuron Model

Now say some arbitrary time-varying current $I_{in}(t)$ is injected into the neuron, be it via electrical stimulation or from other neurons. The total current in the circuit is conserved, so:

$$I_{in}(t) = I_R + I_C$$

From Ohm’s Law, the membrane potential measured between the inside and outside of the neuron $U_{mem}$ is proportional to the current through the resistor:

$$I_R(t) = \frac{U_{mem}(t)}{R}$$

The capacitance is a proportionality constant between the charge stored on the capacitor $Q$ and $U_{mem}(t)$:

$$Q = CU_{mem}(t)$$

The rate of change of charge gives the capacitive current:

$$\frac{dQ}{dt} = I_C(t) = C \frac{dU_{mem}(t)}{dt}$$
Therefore:

\[ I_{\text{in}}(t) = \frac{U_{\text{mem}}(t)}{R} + C \frac{dU_{\text{mem}}(t)}{dt} \]

\[ \Rightarrow RC \frac{dU_{\text{mem}}(t)}{dt} = -U_{\text{mem}}(t) + RI_{\text{in}}(t) \]

The right hand side of the equation is of units [Voltage]. On the left hand side of the equation, the term \( \frac{dU_{\text{mem}}(t)}{dt} \) is of units [Voltage/Time]. To equate it to the left hand side (i.e., voltage), \( RC \) must be of unit [Time]. We refer to \( \tau = RC \) as the time constant of the circuit:

\[ \tau \frac{dU_{\text{mem}}(t)}{dt} = -U_{\text{mem}}(t) + RI_{\text{in}}(t) \]

The passive membrane is therefore described by a linear differential equation.

For a derivative of a function to be of the same form as the original function, i.e., \( \frac{dU_{\text{mem}}(t)}{dt} \propto U_{\text{mem}}(t) \), this implies the solution is exponential with a time constant \( \tau \).

Say the neuron starts at some value \( U_0 \) with no further input, i.e., \( I_{\text{in}}(t) = 0 \). The solution of the linear differential equation is:

\[ U_{\text{mem}}(t) = U_0 e^{-\frac{t}{\tau}} \]

The general solution is shown below.
Optional: Forward Euler Method to Solving the LIF Neuron Model

We managed to find the analytical solution to the LIF neuron, but it is unclear how this might be useful in a neural network. This time, let’s instead use the forward Euler method to solve the previous linear ordinary differential equation (ODE). This approach might seem arduous, but it gives us a discrete, recurrent representation of the LIF neuron. Once we reach this solution, it can be applied directly to a neural network. As before, the linear ODE describing the RC circuit is:

\[ \tau \frac{dU(t)}{dt} = -U(t) + R I_{\text{in}}(t) \]

The subscript from \( U(t) \) is omitted for simplicity.

First, let’s solve this derivative without taking the limit \( \Delta t \to 0 \):

\[ \tau \frac{U(t + \Delta t) - U(t)}{\Delta t} = -U(t) + R I_{\text{in}}(t) \]

For a small enough \( \Delta t \), this gives a good enough approximation of continuous-time integration. Isolating the membrane at the following time step gives:

\[ U(t + \Delta t) = U(t) + \frac{\Delta t}{\tau} (-U(t) + R I_{\text{in}}(t)) \]

The following function represents this equation:

```python
def leaky_integrate_neuron(U, time_step=1e-3, I=0, R=5e7, C=1e-10):
    tau = R*C
    U = U + (time_step/tau)*(-U + I*R)
    return U
```

The default values are set to \( R = 50 M\Omega \) and \( C = 100 pF \) (i.e., \( \tau = 5 ms \)). These are quite realistic with respect to biological neurons.

Now loop through this function, iterating one time step at a time. The membrane potential is initialized at \( U = 0.9 V \), with the assumption that there is no injected input current, \( I_{\text{in}} = 0 A \). The simulation is performed with a millisecond precision \( \Delta t = 1 \times 10^{-3} s \).

```python
num_steps = 100
U = 0.9
U_trace = []  # keeps a record of U for plotting

for step in range(num_steps):
    U_trace.append(U)
    U = leaky_integrate_neuron(U)  # solve next step of U

plot_mem(U_trace, "Leaky Neuron Model")
```
This exponential decay seems to match what we expected!

### 3 Lapicque’s LIF Neuron Model

This similarity between nerve membranes and RC circuits was observed by Louis Lapicque in 1907. He stimulated the nerve fiber of a frog with a brief electrical pulse, and found that neuron membranes could be approximated as a capacitor with a leakage. We pay homage to his findings by naming the basic LIF neuron model in snnTorch after him.

Most of the concepts in Lapicque’s model carry forward to other LIF neuron models. Now it’s time to simulate this neuron using snnTorch.

#### 3.1 Lapicque: Without Stimulus

Instantiate Lapicque’s neuron using the following line of code. R & C are modified to simpler values, while keeping the previous time constant of $\tau = 5 \times 10^{-3}$s.

```python
# Initialize membrane, input, and output
mem = torch.ones(1) * 0.9  # U=0.9 at t=0
```

The neuron model is now stored in lif1. To use this neuron:

**Inputs**
- spk_in: each element of $I_{in}$ is sequentially passed as an input (0 for now)
- mem: the membrane potential, previously $U[t]$, is also passed as input. Initialize it arbitrarily as $U[0] = 0.9$ V.

**Outputs**
- spk_out: output spike $S_{out}[t + \Delta t]$ at the next time step ('1' if there is a spike; '0' if there is no spike)
- mem: membrane potential $U_{mem}[t + \Delta t]$ at the next time step

These all need to be of type `torch.Tensor`. 

(continues on next page)
These values are only for the initial time step \( t = 0 \). To analyze the evolution of \( \text{mem} \) over time, create a list `mem_rec` to record these values at every time step.

```python
# A list to store a recording of membrane potential
mem_rec = [mem]
```

Now it’s time to run a simulation! At each time step, \( \text{mem} \) is updated and stored in `mem_rec`:

```python
# pass updated value of mem and cur_in[step]=0 at every time step
for step in range(num_steps):
    spk_out, mem = lif1(cur_in[step], mem)
    # Store recordings of membrane potential
    mem_rec.append(mem)
    # convert the list of tensors into one tensor
mem_rec = torch.stack(mem_rec)
# pre-defined plotting function
plot_mem(mem_rec, "Lapicque's Neuron Model Without Stimulus")
```

The membrane potential decays over time in the absence of any input stimuli.

### 3.2 Lapicque: Step Input

Now apply a step current \( I_{\text{in}}(t) \) that switches on at \( t = t_0 \). Given the linear first-order differential equation:

\[
\tau \frac{dU_{\text{mem}}}{dt} = -U_{\text{mem}} + R I_{\text{in}}(t),
\]

the general solution is:

\[
U_{\text{mem}}(t) = I_{\text{in}}(t) R + [U_0 - I_{\text{in}}(t_0) R] e^{-\frac{t}{\tau}}
\]

If the membrane potential is initialized to \( U_{\text{mem}}(t = 0) = 0V \), then:

\[
U_{\text{mem}}(t) = I_{\text{in}}(t) R [1 - e^{-\frac{t}{\tau}}]
\]
Based on this explicit time-dependent form, we expect $U_{\text{mem}}$ to relax exponentially towards $I_{\text{in}} R$. Let’s visualize what this looks like by triggering a current pulse of $I_{\text{in}} = 100mA$ at $t_0 = 10ms$.

```python
# Initialize input current pulse
cur_in = torch.cat((torch.zeros(10), torch.ones(190)*0.1), 0) # input current turns on at t=10

# Initialize membrane, output and recordings
mem = torch.zeros(1) # membrane potential of 0 at t=0
spk_out = torch.zeros(1) # neuron needs somewhere to sequentially dump its output spikes
mem_rec = [mem]

This time, the new values of cur_in are passed to the neuron:

```python
num_steps = 200

# pass updated value of mem and cur_in[step] at every time step
for step in range(num_steps):
    spk_out, mem = lif1(cur_in[step], mem)
    mem_rec.append(mem)

# crunch -list- of tensors into one tensor
mem_rec = torch.stack(mem_rec)
plot_step_current_response(cur_in, mem_rec, 10)
```

As $t \to \infty$, the membrane potential $U_{\text{mem}}$ exponentially relaxes to $I_{\text{in}} R$:

```python
>>> print(f"The calculated value of input pulse [A] x resistance [\(\Omega\)] is: \{cur_in[11]*lif1.˓→R\} V")
>>> print(f"The simulated value of steady-state membrane potential is: \{mem_rec[200][0].˓→V\")
```

(continues on next page)
The calculated value of input pulse \([\text{A}] \times \text{resistance} \) is: \(0.5 \text{ V}\)

The simulated value of steady-state membrane potential is: \(0.4999999403953552 \text{ V}\)

Close enough!

### 3.3 Lapicque: Pulse Input

Now what if the step input was clipped at \(t = 30 \text{ms}\)?

```python
# Initialize current pulse, membrane and outputs
cur_in1 = torch.cat((torch.zeros(10), torch.ones(20)*(0.1), torch.zeros(170)), 0)  # input turns on at \(t=10\), off at \(t=30\)
mem = torch.zeros(1)
spk_out = torch.zeros(1)
mem_rec1 = [mem]

# Neuron simulation
for step in range(num_steps):
    spk_out, mem = lif1(cur_in1[step], mem)
    mem_rec1.append(mem)
mem_rec1 = torch.stack(mem_rec1)

plot_current_pulse_response(cur_in1, mem_rec1, "Lapicque's Neuron Model With Input Pulse \(\rightarrow\)", vline1=10, vline2=30)
```

\(U_{\text{mem}}\) rises just as it did for the step input, but now it decays with a time constant of \(\tau\) as in our first simulation.
Let's deliver approximately the same amount of charge $Q = I \times t$ to the circuit in half the time. This means the input current amplitude must be increased by a little, and the time window must be decreased.

```python
# Increase amplitude of current pulse; half the time.
cur_in2 = torch.cat((torch.zeros(10), torch.ones(10)*0.111, torch.zeros(180)), 0)  # input turns on at t=10, off at t=20
mem = torch.zeros(1)
spk_out = torch.zeros(1)
mem_rec2 = [mem]

# neuron simulation
for step in range(num_steps):
    spk_out, mem = lif1(cur_in2[step], mem)
    mem_rec2.append(mem)
mem_rec2 = torch.stack(mem_rec2)
plot_current_pulse_response(cur_in2, mem_rec2, "Lapicque's Neuron Model With Input Pulse: x1/2 pulse width", vline1=10, vline2=20)
```

Let's do that again, but with an even faster input pulse and higher amplitude:

```python
# Increase amplitude of current pulse; quarter the time.
cur_in3 = torch.cat((torch.zeros(10), torch.ones(5)*0.147, torch.zeros(185)), 0)  # input turns on at t=10, off at t=15
mem = torch.zeros(1)
spk_out = torch.zeros(1)
mem_rec3 = [mem]

# neuron simulation
for step in range(num_steps):
    (continues on next page)
```
spk_out, mem = lif1(cur_in3[step], mem)
mem_rec3.append(mem)
mem_rec3 = torch.stack(mem_rec3)

plot_current_pulse_response(cur_in3, mem_rec3, "Lapicque's Neuron Model With Input Pulse: x1/4 pulse width",
                          vline1=10, vline2=15)

Now compare all three experiments on the same plot:

compare_plots(cur_in1, cur_in2, cur_in3, mem_rec1, mem_rec2, mem_rec3, 10, 15,
              20, 30, "Lapicque's Neuron Model With Input Pulse: Varying inputs")
As the input current pulse amplitude increases, the rise time of the membrane potential speeds up. In the limit of the input current pulse width becoming infinitesimally small, $T_W \to 0s$, the membrane potential will jump straight up in virtually zero rise time:

```
# Current spike input
cur_in4 = torch.cat((torch.zeros(10), torch.ones(1)*0.5, torch.zeros(189)), 0)  # input → only on for 1 time step
mem = torch.zeros(1)
spk_out = torch.zeros(1)
mem_rec4 = [mem]

# neuron simulation
for step in range(num_steps):
    spk_out, mem = lif1(cur_in4[step], mem)
    mem_rec4.append(mem)
mem_rec4 = torch.stack(mem_rec4)

plot_current_pulse_response(cur_in4, mem_rec4, "Lapicque's Neuron Model With Input Spike →",
                           vline1=10, ylim_max1=0.6)
```
The current pulse width is now so short, it effectively looks like a spike. That is to say, charge is delivered in an infinitely short period of time, $I_{in}(t) = Q/t_0$ where $t_0 \to 0$. More formally:

$$I_{in}(t) = Q\delta(t - t_0),$$

where $\delta(t - t_0)$ is the Dirac-Delta function. Physically, it is impossible to ‘instantaneously’ deposit charge. But integrating $I_{in}$ gives a result that makes physical sense, as we can obtain the charge delivered:

$$1 = \int_{t_0-a}^{t_0+a} \delta(t - t_0) dt$$

$$f(t_0) = \int_{t_0-a}^{t_0+a} f(t)\delta(t - t_0) dt$$

Here, $f(t_0) = I_{in}(t_0 = 10) = 0.5 A \implies f(t) = Q = 0.5 C$.

Hopefully you have a good feel of how the membrane potential leaks at rest, and integrates the input current. That covers the ‘leaky’ and ‘integrate’ part of the neuron. How about the fire?

### 3.4 Lapicque: Firing

So far, we have only seen how a neuron will react to spikes at the input. For a neuron to generate and emit its own spikes at the output, the passive membrane model must be combined with a threshold.

If the membrane potential exceeds this threshold, then a voltage spike will be generated, external to the passive membrane model.
Modify the `leaky_integrate_neuron` function from before to add a spike response.

```python
# R=5.1, C=5e-3 for illustrative purposes
def leaky_integrate_and_fire(mem, cur=0, threshold=1, time_step=1e-3, R=5.1, C=5e-3):
    tau_mem = R*C
    spk = (mem > threshold) # if membrane exceeds threshold, spk=1, else, 0
    mem = mem + (time_step/tau_mem)*(-mem + cur*R)
    return mem, spk
```

Set `threshold=1`, and apply a step current to get this neuron spiking.

```python
# Small step current input
cur_in = torch.cat((torch.zeros(10), torch.ones(190)*0.2), 0)
mem = torch.zeros(1)
mem_rec = []
spk_rec = []

# neuron simulation
for step in range(num_steps):
    (continues on next page)
```
Oops - the output spikes have gone out of control! This is because we forgot to add a reset mechanism. In reality, each time a neuron fires, the membrane potential hyperpolarizes back to its resting potential.

Implementing this reset mechanism into our neuron:

```python
# LIF w/Reset mechanism
def leaky_integrate_and_fire(mem, cur=0, threshold=1, time_step=1e-3, R=5.1, C=5e-3):
    tau_mem = R*C
    spk = (mem > threshold)
    mem = mem + (time_step/tau_mem)*(-mem + cur*R) - spk*threshold  # every time spk=1, subtract the threshold
    return mem, spk
```

```
# Small step current input
cur_in = torch.cat((torch.zeros(10), torch.ones(190)*0.2), 0)
mem = torch.zeros(1)
mem_rec = []
spk_rec = []
```
# neuron simulation

```python
for step in range(num_steps):
    mem, spk = leaky_integrate_and_fire(mem, cur_in[step])
    mem_rec.append(mem)
    spk_rec.append(spk)

# convert lists to tensors
mem_rec = torch.stack(mem_rec)
spk_rec = torch.stack(spk_rec)

plot_cur_mem_spk(cur_in, mem_rec, spk_rec, thr_line=1, vline=109, ylim_max2=1.3,
                  title="LIF Neuron Model With Reset")
```

Bam. We now have a functional leaky integrate-and-fire neuron model!

Note that if $I_{in} = 0.2 \, \text{A}$ and $R < 5 \, \Omega$, then $I \times R < 1 \, \text{V}$. If threshold=1, then no spiking would occur. Feel free to go back up, change the values, and test it out.

As before, all of that code is condensed by calling the built-in Lapicque neuron model from snnTorch:

```python
# Create the same neuron as before using snnTorch
lif2 = snn.Lapicque(R=5.1, C=5e-3, time_step=1e-3)

>>> print(f"Membrane potential time constant: {lif2.R * lif2.C:.3f}s")
"Membrane potential time constant: 0.025s"

# Initialize inputs and outputs
cur_in = torch.cat((torch.zeros(10), torch.ones(190)*0.2), 0)
mem = torch.zeros(1)
spk_out = torch.zeros(1)
mem_rec = [mem]
```

(continues on next page)
spk_rec = [spk_out]

# Simulation run across 100 time steps.
for step in range(num_steps):
    spk_out, mem = lif2(cur_in[step], mem)
    mem_rec.append(mem)
    spk_rec.append(spk_out)

# convert lists to tensors
mem_rec = torch.stack(mem_rec)
spk_rec = torch.stack(spk_rec)

plot_cur_mem_spk(cur_in, mem_rec, spk_rec, thr_line=1, vline=109, ylim_max2=1.3,
                 title="Lapicque Neuron Model With Step Input")

The membrane potential exponentially rises and then hits the threshold, at which point it resets. We can roughly see this occurs between $105 \text{ ms} < t_{\text{spk}} < 115 \text{ ms}$. As a matter of curiosity, let’s see what the spike recording actually consists of:

```python
>>> print(spk_rec[105:115].view(-1))
tensor([0., 0., 0., 0., 1., 0., 0., 0., 0., 0.])
```

The absence of a spike is represented by $S_{\text{out}} = 0$, and the occurrence of a spike is $S_{\text{out}} = 1$. Here, the spike occurs at $S_{\text{out}}[t = 109] = 1$. If you are wondering why each of these entries is stored as a tensor, it is because in future tutorials we will simulate large scale neural networks. Each entry will contain the spike responses of many neurons, and tensors can be loaded into GPU memory to speed up the training process.

If $I_{\text{in}}$ is increased, then the membrane potential approaches the threshold $U_{\text{thr}}$ faster:

```python
# Initialize inputs and outputs
cur_in = torch.cat(((torch.zeros(10), torch.ones(190)*0.3), 0))  # increased current
```

(continues on next page)
mem = torch.zeros(1)
spk_out = torch.zeros(1)
mem_rec = [mem]
spk_rec = [spk_out]

# neuron simulation
for step in range(num_steps):
    spk_out, mem = lif2(cur_in[step], mem)
    mem_rec.append(mem)
    spk_rec.append(spk_out)

# convert lists to tensors
mem_rec = torch.stack(mem_rec)
spk_rec = torch.stack(spk_rec)

plot_cur_mem_spk(cur_in, mem_rec, spk_rec, thr_line=1, ylim_max2=1.3,
                 title="Lapicque Neuron Model With Periodic Firing")

A similar increase in firing frequency can also be induced by decreasing the threshold. This requires initializing a new neuron model, but the rest of the code block is the exact same as above:

# neuron with halved threshold
lif3 = snn.Lapicque(R=5.1, C=5e-3, time_step=1e-3, threshold=0.5)

# Initialize inputs and outputs
cur_in = torch.cat((torch.zeros(10), torch.ones(190)*0.3), 0)
mem = torch.zeros(1)
spk_out = torch.zeros(1)
mem_rec = [mem]

(continues on next page)
spk_rec = [spk_out]

# Neuron simulation
for step in range(num_steps):
    spk_out, mem = lif3(cur_in[step], mem)
    mem_rec.append(mem)
    spk_rec.append(spk_out)

# convert lists to tensors
mem_rec = torch.stack(mem_rec)
spk_rec = torch.stack(spk_rec)

plot_cur_mem_spk(cur_in, mem_rec, spk_rec, thr_line=0.5, ylim_max2=1.3,
                 title="Lapicque Neuron Model With Lower Threshold")

That’s what happens for a constant current injection. But in both deep neural networks and in the biological brain, most neurons will be connected to other neurons. They are more likely to receive spikes, rather than injections of constant current.

3.5 Lapicque: Spike Inputs

Let’s harness some of the skills we learnt in Tutorial 1, and use the snntorch.spikegen module to create some randomly generated input spikes.

# Create a 1-D random spike train. Each element has a probability of 40% of firing.
spk_in = spikegen.rate_conv(torch.ones((num_steps)) * 0.40)

Run the following code block to see how many spikes have been generated.
```python
>>> print(f"There are \{int(sum(spk_in))\} total spikes out of \{len(spk_in)\} time steps.")
There are 85 total spikes out of 200 time steps.
```

```python
fig = plt.figure(facecolor="w", figsize=(8, 1))
ax = fig.add_subplot(111)
splt.raster(spk_in.reshape(num_steps, -1), ax, s=100, c="black", marker="|")
plt.title("Input Spikes")
plt.xlabel("Time step")
plt.yticks([])
plt.show()
```

```python
# Initialize inputs and outputs
mem = torch.ones(1)*0.5
spk_out = torch.zeros(1)
mem_rec = [mem]
spk_rec = [spk_out]

# Neuron simulation
for step in range(num_steps):
    spk_out, mem = lif3(spk_in[step], mem)
    spk_rec.append(spk_out)
    mem_rec.append(mem)

# convert lists to tensors
mem_rec = torch.stack(mem_rec)
spk_rec = torch.stack(spk_rec)

plot_spk_mem_spk(spk_in, mem_rec, spk_out, "Lapicque’s Neuron Model With Input Spikes")
```
We already implemented a reset mechanism from scratch, but let’s dive a little deeper. This sharp drop of membrane potential promotes a reduction of spike generation, which supplements part of the theory on how brains are so power efficient. Biologically, this drop of membrane potential is known as ‘hyperpolarization’. Following that, it is momentarily more difficult to elicit another spike from the neuron. Here, we use a reset mechanism to model hyperpolarization.

There are two ways to implement the reset mechanism:

1. reset by subtraction (default) — subtract the threshold from the membrane potential each time a spike is generated;
2. reset to zero — force the membrane potential to zero each time a spike is generated.
3. no reset — do nothing, and let the firing go potentially uncontrolled.
Instantiate another neuron model to demonstrate how to alternate between reset mechanisms. By default, snnTorch neuron models use \texttt{reset\_mechanism = "subtract"}. This can be explicitly overridden by passing the argument \texttt{reset\_mechanism = "zero"}.

```python
# Neuron with reset\_mechanism set to "zero"
lif4 = snn.Lapicque(R=5.1, C=5e-3, time\_step=1e-3, threshold=0.5, reset\_mechanism="zero")

# Initialize inputs and outputs
spk\_in = spikegen\_rate\_conv(torch.\ones((num\_steps)) \* 0.4)  
mem = torch.\ones(1)\*0.5  
spk\_out = torch.\zeros(1)  
mem\_rec0 = [mem]  
spk\_rec0 = [spk\_out]

# Neuron simulation
for step in range(num\_steps):
    spk\_out, mem = lif4(spk\_in[step], mem)  
    spk\_rec0.\append(spk\_out)  
    mem\_rec0.\append(mem)

# convert lists to tensors
mem\_rec0 = torch.\stack(mem\_rec0)  
spk\_rec0 = torch.\stack(spk\_rec0)

plot\_reset\_comparison(spk\_in, mem\_rec, spk\_rec, mem\_rec0, spk\_rec0)
```
Pay close attention to the evolution of the membrane potential, especially in the moments after it reaches the threshold. You may notice that for “Reset to Zero”, the membrane potential is forced back to zero after each spike.

So which one is better? Applying "subtract" (the default value in \texttt{reset\_mechanism}) is less lossy, because it does not ignore how much the membrane exceeds the threshold by.

On the other hand, applying a hard reset with "zero" promotes sparsity and potentially less power consumption when running on dedicated neuromorphic hardware. Both options are available for you to experiment with.

That covers the basics of a LIF neuron model!

\section*{Conclusion}

In practice, we probably wouldn’t use this neuron model to train a neural network. The Lapicque LIF model has added a lot of hyperparameters to tune: $R$, $C$, $\Delta t$, $U_{\text{thr}}$, and the choice of reset mechanism. It’s all a little bit daunting. So the next tutorial will eliminate most of these hyperparameters, and introduce a neuron model that is better suited for large-scale deep learning.

If you like this project, please consider starring the repo on GitHub as it is the easiest and best way to support it.

For reference, the documentation can be found here.

\section*{Further Reading}

- Check out the \texttt{snnTorch} GitHub project here.
- \texttt{snnTorch} documentation of the Lapicque, Leaky, Synaptic, and Alpha models
- \textit{Neuronal Dynamics: From single neurons to networks and models of cognition} by Wulfram Gerstner, Werner M. Kistler, Richard Naud and Liam Paninski.
- \textit{Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems} by Laurence F. Abbott and Peter Dayan
Tutorial 3 - A Feedforward Spiking Neural Network

Tutorial written by Jason K. Eshraghian (www.ncg.ucsc.edu)

The snnTorch tutorial series is based on the following paper. If you find these resources or code useful in your work, please consider citing the following source:


Note:
This tutorial is a static non-editable version. Interactive, editable versions are available via the following links:

- Google Colab
- Local Notebook (download via GitHub)

Introduction

In this tutorial, you will:

- Learn how to simplify the leaky integrate-and-fire (LIF) neuron to make it deep learning-friendly
- Implement a feedforward spiking neural network (SNN)

Install the latest PyPi distribution of snnTorch:

```
$ pip install snntorch
```

```python
# imports
import snntorch as snn
from snntorch import spikeplot as splt
from snntorch import spikegen

import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```

1. Simplifying the Leaky Integrate-and-Fire Neuron Model

In the previous tutorial, we designed our own LIF neuron model. But it was quite complex, and added an array of hyperparameters to tune, including $R$, $C$, $\Delta t$, $U_{\text{thr}}$, and the choice of reset mechanism. This is a lot to keep track of, and only grows more cumbersome when scaled up to full-blown SNN. So let’s make a few simplifications.
1.1 The Decay Rate: beta

In the previous tutorial, the Euler method was used to derive the following solution to the passive membrane model:

\[ U(t + \Delta t) = (1 - \frac{\Delta t}{\tau})U(t) + \frac{\Delta t}{\tau} I_{in}(t)R \]

Now assume there is no input current, \( I_{in}(t) = 0 \):

\[ U(t + \Delta t) = (1 - \frac{\Delta t}{\tau})U(t) \]

Let the ratio of subsequent values of \( U \), i.e., \( U(t + \Delta t)/U(t) \) be the decay rate of the membrane potential, also known as the inverse time constant:

\[ U(t + \Delta t) = \beta U(t) \]

From (1), this implies that:

\[ \beta = (1 - \frac{\Delta t}{\tau}) \]

For reasonable accuracy, \( \Delta t << \tau \).

1.2 Weighted Input Current

If we assume \( t \) represents time-steps in a sequence rather than continuous time, then we can set \( \Delta t = 1 \). To further reduce the number of hyperparameters, assume \( R = 1 \). From (4), these assumptions lead to:

\[ \beta = (1 - \frac{1}{C}) \implies (1 - \beta)I_{in} = \frac{1}{\tau}I_{in} \]

The input current is weighted by \( (1 - \beta) \). By additionally assuming input current instantaneously contributes to the membrane potential:

\[ U[t + 1] = \beta U[t] + (1 - \beta)I_{in}[t + 1] \]

Note that the discretization of time means we are assuming that each time bin \( t \) is brief enough such that a neuron may only emit a maximum of one spike in this interval.

In deep learning, the weighting factor of an input is often a learnable parameter. Taking a step away from the physically viable assumptions made thus far, we subsume the effect of \( (1 - \beta) \) from (6) into a learnable weight \( W \), and replace \( I_{in}[t] \) accordingly with an input \( X[t] \):

\[ WX[t] = I_{in}[t] \]

This can be interpreted in the following way. \( X[t] \) is an input voltage, or spike, and is scaled by the synaptic conductance of \( W \) to generate a current injection to the neuron. This gives us the following result:

\[ U[t + 1] = \beta U[t] + WX[t + 1] \]

In future simulations, the effects of \( W \) and \( \beta \) are decoupled. \( W \) is a learnable parameter that is updated independently of \( \beta \).
1.3 Spiking and Reset

We now introduce the spiking and reset mechanisms. Recall that if the membrane exceeds the threshold, then the neuron emits an output spike:

\[ S[t] = \begin{cases} 1, & \text{if } U[t] > U_{\text{thr}} \\ 0, & \text{otherwise} \end{cases} \]

If a spike is triggered, the membrane potential should be reset. The reset-by-subtraction mechanism is modeled by:

\[ U[t + 1] = \beta U[t] + W X[t + 1] - S[t] U_{\text{thr}} \]

As \( W \) is a learnable parameter, and \( U_{\text{thr}} \) is often just set to 1 (though can be tuned), this leaves the decay rate \( \beta \) as the only hyperparameter left to be specified. This completes the painful part of this tutorial.

**Note:** Some implementations might make slightly different assumptions. E.g., \( S[t] \rightarrow S[t + 1] \) in (9), or \( X[t] \rightarrow X[t + 1] \) in (10). This above derivation is what is used in snnTorch as we find it maps intuitively to a recurrent neural network representation, without any change in performance.

1.4 Code Implementation

Implementing this neuron in Python looks like this:

```python
def leaky_integrate_and_fire(mem, x, w, beta, threshold=1):
    spk = (mem > threshold)  # if membrane exceeds threshold, spk=1, else, 0
    mem = beta * mem + w * x - spk * threshold
    return spk, mem
```

To set \( \beta \), we have the option of either using Equation (3) to define it, or hard-coding it directly. Here, we will use (3) for the sake of a demonstration, but in future, it will just be hard-coded as we are more focused on something that works rather than biological precision.

Equation (3) tells us that \( \beta \) is the ratio of membrane potential across two subsequent time steps. Solve this using the continuous time-dependent form of the equation (assuming no current injection), which was derived in Tutorial 2:

\[ U(t) = U_0 e^{-\frac{t}{\tau}} \]

\( U_0 \) is the initial membrane potential at \( t = 0 \). Assume the time-dependent equation is computed at discrete steps of \( t, (t + \Delta t), (t + 2\Delta t) \ldots \), then we can find the ratio of membrane potential between subsequent steps using:

\[ \beta = \frac{U_0 e^{-\frac{t + \Delta t}{\tau}}}{U_0 e^{-\frac{t}{\tau}}} = \frac{U_0 e^{-\frac{t + 2\Delta t}{\tau}}}{U_0 e^{-\frac{t}{\tau}}} = \ldots \]

\[ \Rightarrow \beta = e^{-\frac{\Delta t}{\tau}} \]

```python
# set neuronal parameters
delta_t = torch.tensor(1e-3)
tau = torch.tensor(5e-3)
beta = torch.exp(-delta_t/tau)
```
Run a quick simulation to check the neuron behaves correctly in response to a step voltage input:

```python
num_steps = 200

# initialize inputs/outputs + small step current input
x = torch.cat((torch.zeros(10), torch.ones(190)*0.5), 0)
mem = torch.zeros(1)
spk_out = torch.zeros(1)
mem_rec = []
spk_rec = []

# neuron parameters
w = 0.4
beta = 0.819

# neuron simulation
for step in range(num_steps):
    spk, mem = leaky_integrate_and_fire(mem, x[step], w=w, beta=beta)
    mem_rec.append(mem)
    spk_rec.append(spk)

# convert lists to tensors
mem_rec = torch.stack(mem_rec)
spk_rec = torch.stack(spk_rec)

plot_cur_mem_spk(x*w, mem_rec, spk_rec, thr_line=1, ylim_max1=0.5,
                 title="LIF Neuron Model With Weighted Step Voltage")
```
2. Leaky Neuron Model in snnTorch

This same thing can be achieved by instantiating `snn.Leaky`, in a similar way to how we used `snn.Lapicque` in the previous tutorial, but with less hyperparameters:

```python
lif1 = snn.Leaky(beta=0.8)
```

The neuron model is now stored in `lif1`. To use this neuron:

**Inputs**
- `cur_in`: each element of $W \times X[t]$ is sequentially passed as an input
- `mem`: the previous step membrane potential, $U[t-1]$, is also passed as input.

**Outputs**
- `spk_out`: output spike $S[t]$ (‘1’ if there is a spike; ‘0’ if there is no spike)
- `mem`: membrane potential $U[t]$ of the present step

These all need to be of type `torch.Tensor`. Note that here, we assume the input current has already been weighted before passing into the `snn.Leaky` neuron. This will make more sense when we construct a network-scale model. Also, equation (10) has been time-shifted back one step without loss of generality.

```python
# Small step current input
w=0.21
cur_in = torch.cat((torch.zeros(10), torch.ones(190)*w), 0)
mem = torch.zeros(1)
spk = torch.zeros(1)
mem_rec = []
spk_rec = []

# neuron simulation
for step in range(num_steps):
    spk, mem = lif1(cur_in[step], mem)
    mem_rec.append(mem)
    spk_rec.append(spk)

# convert lists to tensors
mem_rec = torch.stack(mem_rec)
spk_rec = torch.stack(spk_rec)

plot_cur_mem_spk(cur_in, mem_rec, spk_rec, thr_line=1, ylim_max1=0.5,
                   title="snn.Leaky Neuron Model")
```

This model has the same optional input arguments of `reset_mechanism` and `threshold` as described for Lapicque's neuron model.
3. A Feedforward Spiking Neural Network

So far, we have only considered how a single neuron responds to input stimulus. snnTorch makes it straightforward to scale this up to a deep neural network. In this section, we will create a 3-layer fully-connected neural network of dimensions 784-1000-10. Compared to our simulations so far, each neuron will now integrate over many more incoming input spikes.
PyTorch is used to form the connections between neurons, and snnTorch to create the neurons. First, initialize all layers.

```python
# layer parameters
num_inputs = 784
num_hidden = 1000
num_outputs = 10
beta = 0.99

# initialize layers
fc1 = nn.Linear(num_inputs, num_hidden)
lif1 = snn.Leaky(beta=beta)
fc2 = nn.Linear(num_hidden, num_outputs)
lif2 = snn.Leaky(beta=beta)
```

Next, initialize the hidden variables and outputs of each spiking neuron. As networks increase in size, this becomes more tedious. The static method `init_leaky()` can be used to take care of this. All neurons in snnTorch have their own initialization methods that follow this same syntax, e.g., `init_lapicque()`. The shape of the hidden states are automatically initialized based on the input data dimensions during the first forward pass.
# Initialize hidden states
mem1 = lif1.init_leaky()
mem2 = lif2.init_leaky()

# record outputs
mem2_rec = []
spk1_rec = []
spk2_rec = []

Create an input spike train to pass to the network. There are 200 time steps to simulate across 784 input neurons, i.e.,
the input originally has dimensions of 200 × 784. However, neural nets typically process data in minibatches. snnTorch,
uses time-first dimensionality:

\[ [\text{time} \times \text{batch size} \times \text{feature dimensions}] \]

So ‘unsqueeze’ the input along `dim=1` to indicate ‘one batch’ of data. The dimensions of this input tensor must be 200
× 1 × 784:

```python
spk_in = spikegen.rate_conv(torch.rand((200, 784))).unsqueeze(1)
>>> print(f"Dimensions of spk_in: \{spk_in.size()\}")
"Dimensions of spk_in: torch.Size([200, 1, 784])"
```

Now it’s finally time to run a full simulation. An intuitive way to think about how PyTorch and snnTorch work together
is that PyTorch routes the neurons together, and snnTorch loads the results into spiking neuron models. In terms of
coding up a network, these spiking neurons can be treated like time-varying activation functions.

Here is a sequential account of what’s going on:

- The \( i^{th} \) input from \( \text{spk\textunderscore in} \) to the \( j^{th} \) neuron is weighted by the parameters initialized in \texttt{nn.Linear}: \( X_i \times W_{ij} \)
- This generates the input current term from Equation (10), contributing to \( U[t+1] \) of the spiking neuron
- If \( U[t+1] > U_{\text{thr}} \), then a spike is triggered from this neuron
- This spike is weighted by the second layer weight, and the above process is repeated for all inputs, weights, and
neurons.
- If there is no spike, then nothing is passed to the post-synaptic neuron.

The only difference from our simulations thus far is that we are now scaling the input current with a weight generated
by \texttt{nn.Linear}, rather than manually setting \( W \) ourselves.

```python
# network simulation
for step in range(num_steps):
    cur1 = fc1(spk_in[step])  # post-synaptic current <-- spk_in x weight
    spk1, mem1 = lif1(cur1, mem1)  # mem[t+1] <-- post-syn current + decayed membrane
    cur2 = fc2(spk1)
    spk2, mem2 = lif2(cur2, mem2)

    mem2_rec.append(mem2)
    spk1_rec.append(spk1)
    spk2_rec.append(spk2)

# convert lists to tensors
mem2_rec = torch.stack(mem2_rec)
spk1_rec = torch.stack(spk1_rec)
spk2_rec = torch.stack(spk2_rec)
```

(continues on next page)
At this stage, the spikes don’t have any real meaning. The inputs and weights are all randomly initialized, and no training has taken place. But the spikes should appear to be propagating from the first layer through to the output. If you are not seeing any spikes, then you might have been unlucky in the weight initialization lottery - you might want to try re-running the last four code-blocks.

`snnplot.spike_count` can create a spike counter of the output layer. The following animation will take some time to generate.

Note: if you are running the notebook locally on your desktop, please uncomment the line below and modify the path to your ffmpeg.exe

```python
from IPython.display import HTML

fig, ax = plt.subplots(facecolor='w', figsize=(12, 7))
labels=['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
spk2_rec = spk2_rec.squeeze(1).detach().cpu()

# plt.rcParams['animation.ffmpeg_path'] = 'C:\path\to\your\ffmpeg.exe'

# Plot spike count histogram
anim = splt.spike_count(spk2_rec, fig, ax, labels=labels, animate=True)
HTML(anim.to_html5_video())
```

1.11. License & Copyright
spikeplot.traces lets you visualize the membrane potential traces. We will plot 9 out of 10 output neurons. Compare it to the animation and raster plot above to see if you can match the traces to the neuron.

```python
# plot membrane potential traces
splt.traces(mem2_rec.squeeze(1), spk=spk2_rec.squeeze(1))
fig = plt.gcf()
fig.set_size_inches(8, 6)
```

It is fairly normal if some neurons are firing while others are completely dead. Again, none of these spikes have any real meaning until the weights have been trained.

**Conclusion**

That covers how to simplify the leaky integrate-and-fire neuron model, and then using it to build a spiking neural network. In practice, we will almost always prefer to use `snn.Leaky` over `snn.Lapicque` for training networks, as there is a smaller hyperparameter search space.

Tutorial 4 goes into detail with the 2nd-order `snn.Synaptic` and `snn.Alpha` models. This next tutorial is not necessary for training a network, so if you wish to go straight to deep learning with snnTorch, then skip ahead to Tutorial 5.

If you like this project, please consider starring the repo on GitHub as it is the easiest and best way to support it. For reference, the documentation can be found here.
Further Reading

- Check out the snnTorch GitHub project here.
- snnTorch documentation of the Lapicque, Leaky, Synaptic, and Alpha models
- Neuronal Dynamics: From single neurons to networks and models of cognition by Wulfram Gerstner, Werner M. Kistler, Richard Naud and Liam Paninski.
- Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems by Laurence F. Abbott and Peter Dayan

Tutorial 4 - 2nd Order Spiking Neuron Models

Tutorial written by Jason K. Eshraghian (www.ncg.ucsc.edu)

The snnTorch tutorial series is based on the following paper. If you find these resources or code useful in your work, please consider citing the following source:


Note:
This tutorial is a static non-editable version. Interactive, editable versions are available via the following links:
- Google Colab
- Local Notebook (download via GitHub)

Introduction

In this tutorial, you will:
- Learn about the more advanced leaky integrate-and-fire (LIF) neuron models available: Synaptic and Alpha

Install the latest PyPi distribution of snnTorch.

```
$ pip install snntorch
```

```
# imports
import snntorch as snn
from snntorch import spikeplot as splt
from snntorch import spikegen

import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```
1. Synaptic Conductance-based LIF Neuron Model

The neuron models explored in previous tutorials assume that an input voltage spike leads to an instantaneous jump in synaptic current, which then contributes to the membrane potential. In reality, a spike will result in the gradual release of neurotransmitters from the pre-synaptic neuron to the post-synaptic neuron. The synaptic conductance-based LIF model accounts for the gradual temporal dynamics of input current.

1.1 Modeling Synaptic Current

If a pre-synaptic neuron fires, the voltage spike is transmitted down the axon of the neuron. It triggers the vesicles to release neurotransmitters into the synaptic cleft. These activate the post-synaptic receptors, which directly influence the effective current that flows into the post-synaptic neuron. Shown below are two types of excitatory receptors, AMPA and NMDA.

The simplest model of synaptic current assumes an increasing current on a very fast time-scale, followed by a relatively slow exponential decay, as seen in the AMPA receptor response above. This is very similar to the membrane potential dynamics of Lapicque’s model.
The synaptic model has two exponentially decaying terms: $I_{\text{syn}}(t)$ and $U_{\text{mem}}(t)$. The ratio between subsequent terms (i.e., decay rate) of $I_{\text{syn}}(t)$ is set to $\alpha$, and that of $U(t)$ is set to $\beta$:

$$\alpha = e^{-\Delta t/\tau_{\text{syn}}}$$

$$\beta = e^{-\Delta t/\tau_{\text{mem}}}$$

where the duration of a single time step is normalized to $\Delta t = 1$ in future. $\tau_{\text{syn}}$ models the time constant of the synaptic current in an analogous way to how $\tau_{\text{mem}}$ models the time constant of the membrane potential. $\beta$ is derived in the exact same way as the previous tutorial, with a similar approach to $\alpha$:

$$I_{\text{syn}}[t + 1] = \alpha I_{\text{syn}}[t] + W X[t + 1]$$

$$U[t + 1] = \beta U[t] + I_{\text{syn}}[t + 1] - R[t]$$

The same conditions for spiking as the previous LIF neurons still hold:

$$S_{\text{out}}[t] = \begin{cases} 1, & \text{if } U[t] > U_{\text{thr}} \\ 0, & \text{otherwise} \end{cases}$$

### 1.2 Synaptic Neuron Model in snnTorch

The synaptic conductance-based neuron model combines the synaptic current dynamics with the passive membrane. It must be instantiated with two input arguments:

- $\alpha$: the decay rate of the synaptic current
- $\beta$: the decay rate of the membrane potential (as with Lapicque)

```python
# Temporal dynamics
alpha = 0.9
beta = 0.8
num_steps = 200

# Initialize 2nd-order LIF neuron
lif1 = snn.Synaptic(alpha=alpha, beta=beta)
```

Using this neuron is the exact same as previous LIF neurons, but now with the addition of synaptic current syn as an input and output:

**Inputs**

- spk_in: each weighted input voltage spike $WX[t]$ is sequentially passed in
- syn: synaptic current $I_{\text{syn}}[t - 1]$ at the previous time step
- mem: membrane potential $U[t - 1]$ at the previous time step

**Outputs**

- spk_out: output spike $S[t]$ (‘1’ if there is a spike; ‘0’ if there is no spike)
- syn: synaptic current $I_{\text{syn}}[t]$ at the present time step
- mem: membrane potential $U[t]$ at the present time step
These all need to be of type `torch.Tensor`. Note that the neuron model has been time-shifted back one step without loss of generality.

Apply a periodic spiking input to see how current and membrane evolve with time:

```python
# Periodic spiking input, spk_in = 0.2 V
w = 0.2
spk_period = torch.cat((torch.ones(1)*w, torch.zeros(9)), 0)
spk_in = spk_period.repeat(20)

# Initialize hidden states and output
syn, mem = lif1.init_synaptic()
spk_out = torch.zeros(1)
syn_rec = []
mem_rec = []
spk_rec = []

# Simulate neurons
for step in range(num_steps):
    spk_out, syn, mem = lif1(spk_in[step], syn, mem)
    spk_rec.append(spk_out)
    syn_rec.append(syn)
    mem_rec.append(mem)

# convert lists to tensors
spk_rec = torch.stack(spk_rec)
syn_rec = torch.stack(syn_rec)
mem_rec = torch.stack(mem_rec)

plot_spk_cur_mem_spk(spk_in, syn_rec, mem_rec, spk_rec,
                     "Synaptic Conductance-based Neuron Model With Input Spikes")
```
This model also has the optional input arguments of `reset_mechanism` and `threshold` as described for Lapicque’s neuron model. In summary, each spike contributes a shifted exponential decay to the synaptic current $I_{\text{syn}}$, which are all summed together. This current is then integrated by the passive membrane equation derived in Tutorial 2, thus generating output spikes. An illustration of this process is provided below.

![Synaptic Conductance-based Neuron Model With Input Spikes](image1)

- **Synaptic Conductance-based Neuron Model With Input Spikes**
  - Input spikes
  - Synaptic Current ($I_{\text{syn}}$)
  - Membrane Potential ($U_{\text{mem}}$)
  - Output spikes

![A Graphical Approach to Modelling Synaptic Current](image2)

- **A Graphical Approach to Modelling Synaptic Current**
  - $S_{\text{in}}$ to $S_{\text{out}}$
  - $I_{\text{syn}}$
  - $U$
  - $U_{\text{thr}}$

- Synaptic current integrates input spikes and decays at a rate of $\alpha$.
- Membrane potential integrates the synaptic current and decays at a rate of $\beta$. An output spike is generated each time the membrane crosses the threshold.
1.3 1st-Order vs. 2nd-Order Neurons

A natural question that arises is - *when do I want to use a 1st order LIF neuron and when should I use this 2nd order LIF neuron?* While this has not really been settled, my own experiments have given me some intuition that might be useful.

**When 2nd-order neurons are better**

- If the temporal relations of your input data occur across long time-scales,
- or if the input spiking pattern is sparse

By having two recurrent equations with two decay terms ($\alpha$ and $\beta$), this neuron model is able to ‘sustain’ input spikes over a longer duration. This can be beneficial to retaining long-term relationships.

An alternative use case might also be:

- When temporal codes matter

If you care for the precise timing of a spike, it seems easier to control that for a 2nd-order neuron. In the *Leaky* model, a spike would be triggered in direct synchrony with the input. For 2nd-order models, the membrane potential is ‘smoothed out’ (i.e., the synaptic current model low-pass filters the membrane potential), which means one can use a finite rise time for $U[t]$. This is clear in the previous simulation, where the output spikes experience a delay with respect to the input spikes.

**When 1st-order neurons are better**

- Any case that doesn’t fall into the above, and sometimes, the above cases.

By having one less equation in 1st-order neuron models (such as *Leaky*), the backpropagation process is made a little simpler. Though having said that, the *Synaptic* model is functionally equivalent to the *Leaky* model for $\alpha = 0$. In my own hyperparameter sweeps on simple datasets, the optimal results seem to push $\alpha$ as close to 0 as possible. As data increases in complexity, $\alpha$ may grow larger.

2. Alpha Neuron Model (Hacked Spike Response Model)

A recursive version of the Spike Response Model (SRM), or the ‘Alpha’ neuron, is also available, called using `snn.Alpha`. The neuron models thus far have all been based on the passive membrane model, using ordinary differential equations to describe their dynamics.

The SRM family of models, on the other hand, is interpreted in terms of a filter. Upon the arrival of an input spike, this spike is convolved with the filter to give the membrane potential response. The form of this filter can be exponential, as is the case with Lapicque’s neuron, or they can be more complex such as a sum of exponentials. SRM models are appealing as they can arbitrarily add refractoriness, threshold adaptation, and any number of other features simply by embedding them into the filter.
2.1 Modelling the Alpha Neuron Model

Formally, this process is represented by:

\[ U_{\text{mem}}(t) = \sum_{i} W(\epsilon \ast S_{\text{in}})(t) \]

where the incoming spikes \( S_{\text{in}} \) are convolved with a spike response kernel \( \epsilon(\cdot) \). The spike response is scaled by a synaptic weight, \( W \). In top figure, the kernel is an exponentially decaying function and would be the equivalent of Lapicque’s 1st-order neuron model. On the bottom, the kernel is an alpha function:

\[ \epsilon(t) = \frac{t}{\tau} e^{1-t/\tau} \Theta(t) \]

where \( \tau \) is the time constant of the alpha kernel and \( \Theta \) is the Heaviside step function. Most kernel-based methods adopt the alpha function as it provides a time-delay that is useful for temporal codes that are concerned with specifying the exact spike time of a neuron.

In snnTorch, the spike response model is not directly implemented as a filter. Instead, it is recast into a recursive form such that only the previous time step of values are required to calculate the next set of values. This reduces the memory required.

As the membrane potential is now determined by the sum of two exponentials, each of these exponents has its own independent decay rate. \( \alpha \) defines the decay rate of the positive exponential, and \( \beta \) defines the decay rate of the negative exponential.

\[
\begin{align*}
\alpha &= 0.8 \\
\beta &= 0.7
\end{align*}
\]

```
# initialize neuron
lif2 = snn.Alpha(alpha=alpha, beta=beta, threshold=0.5)
```

Using this neuron is the same as the previous neurons, but the sum of two exponential functions requires the synaptic current \( \text{syn} \) to be split into a \( \text{syn}_{\text{exc}} \) and \( \text{syn}_{\text{inh}} \) component:

**Inputs**

- **spk_in**: each weighted input voltage spike \( WX[t] \) is sequentially passed in
- **syn_exc**: excitatory post-synaptic current \( I_{\text{syn}_{\text{exc}}}[t-1] \) at the previous time step
- **syn_inh**: inhibitory post-synaptic current \( I_{\text{syn}_{\text{inh}}}[t-1] \) at the previous time step
• **mem**: membrane potential $U_{\text{mem}}[t-1]$ at the present time $t$ at the previous time step

**Outputs**

• **spk_out**: output spike $S_{\text{out}}[t]$ at the present time step (‘1’ if there is a spike; ‘0’ if there is no spike)
• **syn_exc**: excitatory post-synaptic $I_{\text{syn-exc}}[t]$ at the present time step $t$
• **syn_inh**: inhibitory post-synaptic current $I_{\text{syn-inh}}[t]$ at the present time step $t$
• **mem**: membrane potential $U_{\text{mem}}[t]$ at the present time step

As with all other neuron models, these must be of type `torch.Tensor`.

```python
# input spike: initial spike, and then period spiking
w = 0.85
spk_in = (torch.cat((torch.zeros(10), torch.ones(1), torch.zeros(89),
               (torch.cat((torch.ones(1), torch.zeros(9)),0).repeat(10))), 0) * w).
               --> unsqueeze(1)

# initialize parameters
syn_exc, syn_inh, mem = lif2.init_alpha()
mem_rec = []
spk_rec = []

# run simulation
for step in range(num_steps):
    spk_out, syn_exc, syn_inh, mem = lif2(spk_in[step], syn_exc, syn_inh, mem)
    mem_rec.append(mem.squeeze(0))
    spk_rec.append(spk_out.squeeze(0))

# convert lists to tensors
mem_rec = torch.stack(mem_rec)
spk_rec = torch.stack(spk_rec)
plot_spk_mem_spk(spk_in, mem_rec, spk_rec, "Alpha Neuron Model With Input Spikes")
```
As with the Lapicque and Synaptic models, the Alpha model also has options to modify the threshold and reset mechanism.

### 2.2 Practical Considerations

As mentioned for the Synaptic neuron, the more complex a model, the more complex the backpropagation process during training. In my own experiments, I have yet to find a case where the Alpha neuron outperforms the Synaptic and Leaky neuron models. It seems as though learning through a positive and negative exponential only makes the gradient calculation process more difficult, and offsets any potential benefits in more complex neuronal dynamics.

However, when an SRM model is expressed as a time-varying kernel (rather than a recursive model as is done here), it seems to perform just as well as the simpler neuron models. As an example, see the following paper:


The Alpha neuron has been included with the intent of providing an option for porting across SRM-based models over into snnTorch, although natively training them seems to not be too effective in snnTorch.
Conclusion

We have covered all LIF neuron models available in snnTorch. As a quick summary:

- **Lapicque**: a physically accurate model based directly on RC-circuit parameters
- **Leaky**: a simplified 1st-order model
- **Synaptic**: a 2nd-order model that accounts for synaptic current evolution
- **Alpha**: a 2nd-order model where the membrane potential tracks an alpha function

In general, **Leaky** and **Synaptic** seem to be the most useful for training a network. **Lapicque** is good for demonstrating physically precise models, while **Alpha** is only intended to capture the behaviour of SRM neurons.

Building a network using these slightly more advanced neurons follows the exact same procedure as in Tutorial 3.

If you like this project, please consider starring the repo on GitHub as it is the easiest and best way to support it.

For reference, the documentation can be found here.

Further Reading

- Check out the snnTorch GitHub project here.
- [snnTorch documentation](https://github.com/snnTorch/snnTorch) of the Lapicque, Leaky, Synaptic, and Alpha models
- Neuronal Dynamics: From single neurons to networks and models of cognition by Wulfram Gerstner, Werner M. Kistler, Richard Naud and Liam Paninski.
- Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems by Laurence F. Abbott and Peter Dayan

**Tutorial 5 - Training Spiking Neural Networks with snntorch**

Tutorial written by Jason K. Eshraghian (www.ncg.ucsc.edu)

The snnTorch tutorial series is based on the following paper. If you find these resources or code useful in your work, please consider citing the following source:


Note:

This tutorial is a static non-editable version. Interactive, editable versions are available via the following links:

- Google Colab
- Local Notebook (download via GitHub)
**Introduction**

In this tutorial, you will:

- Learn how spiking neurons are implemented as a recurrent network
- Understand backpropagation through time, and the associated challenges in SNNs such as the non-differentiability of spikes
- Train a fully-connected network on the static MNIST dataset


At the end of the tutorial, a basic supervised learning algorithm will be implemented. We will use the original static MNIST dataset and train a multi-layer fully-connected spiking neural network using gradient descent to perform image classification.

Install the latest PyPi distribution of snnTorch:

```bash
$ pip install snntorch
```

### 1. A Recurrent Representation of SNNs

In Tutorial 3, we derived a recursive representation of a leaky integrate-and-fire (LIF) neuron:

\[
U[t + 1] = \beta U[t] + WX[t + 1] - R[t]
\]

where input synaptic current is interpreted as \(I_{in}[t] = WX[t]\), and \(X[t]\) may be some arbitrary input of spikes, a step/time-varying voltage, or unweighted step/time-varying current. Spiking is represented with the following equation, where if the membrane potential exceeds the threshold, a spike is emitted:

\[
S[t] = \begin{cases} 
1, & \text{if } U[t] > U_{\text{thr}} \\
0, & \text{otherwise} 
\end{cases}
\]

This formulation of a spiking neuron in a discrete, recursive form is almost perfectly poised to take advantage of the developments in training recurrent neural networks (RNNs) and sequence-based models. This is illustrated using an
implicit recurrent connection for the decay of the membrane potential, and is distinguished from explicit recurrence where the output spike $S_{\text{out}}$ is fed back to the input. In the figure below, the connection weighted by $-U_{\text{thr}}$ represents the reset mechanism $R[t]$.

The benefit of an unrolled graph is that it provides an explicit description of how computations are performed. The process of unfolding illustrates the flow of information forward in time (from left to right) to compute outputs and losses, and backward in time to compute gradients. The more time steps that are simulated, the deeper the graph becomes.

Conventional RNNs treat $\beta$ as a learnable parameter. This is also possible for SNNs, though by default, they are treated as hyperparameters. This replaces the vanishing and exploding gradient problems with a hyperparameter search. A future tutorial will describe how to make $\beta$ a learnable parameter.

2. The Non-Differentiability of Spikes

2.1 Training Using the Backprop Algorithm

An alternative way to represent the relationship between $S$ and $U$ in (2) is:

$$S[t] = \Theta(U[t] - U_{\text{thr}})$$

where $\Theta(\cdot)$ is the Heaviside step function:
Training a network in this form poses some serious challenges. Consider a single, isolated time step of the computational graph from the previous figure titled “Recurrent representation of spiking neurons”, as shown in the forward pass below:

The goal is to train the network using the gradient of the loss with respect to the weights, such that the weights are

The derivative of the Heaviside function is 0 almost everywhere, causing $\frac{\partial L}{\partial W}$ to also become 0.
updated to minimize the loss. The backpropagation algorithm achieves this using the chain rule:

\[
\frac{\partial L}{\partial W} = \frac{\partial L}{\partial S} \frac{\partial S}{\partial U} \frac{\partial U}{\partial I} \frac{\partial I}{\partial W} \{0, \infty\}
\]

From (1), \(\partial I/\partial W = X\), and \(\partial U/\partial I = 1\). While a loss function is yet to be defined, we can assume \(\partial L/\partial S\) has an analytical solution, in a similar form to the cross-entropy or mean-square error loss (more on that shortly).

However, the term that we are going to grapple with is \(\partial S/\partial U\). The derivative of the Heaviside step function from (3) is the Dirac Delta function, which evaluates to 0 everywhere, except at the threshold \(U_{\text{thr}} = \theta\), where it tends to infinity. This means the gradient will almost always be nulled to zero (or saturated if \(U\) sits precisely at the threshold), and no learning can take place. This is known as the dead neuron problem.

### 2.2 Overcoming the Dead Neuron Problem

The most common way to address the dead neuron problem is to keep the Heaviside function as it is during the forward pass, but swap the derivative term \(\partial S/\partial U\) for something that does not kill the learning process during the backward pass, which will be denoted \(\partial S^\sim/\partial U\). This might sound odd, but it turns out that neural networks are quite robust to such approximations. This is commonly known as the surrogate gradient approach.

A variety of options exist to using surrogate gradients, and we will dive into more detail on these methods in Tutorial 6. For now, a simple approximation is applied where \(\partial S^\sim/\partial U\) is set to \(S\) itself.

If \(S\) does not spike, then the spike-gradient term is 0. If \(S\) spikes, then the gradient term is 1. This simply looks like the gradient of a ReLU function shifted to the threshold. This method is known as the Spike-Operator approach and is described in more detail in the following paper:


Intuitively, Spike Operator splits the gradient calculation into two chunks: one where the neuron is spiking, and one where it is silent:

- **Silent**: If the neuron is silent, then the spike response can be obtained by scaling the membrane by 0: \(S = U \times 0 \Rightarrow \partial S^\sim/\partial U = 0\).

- **Spiking**: If the neuron is spiking, then assume \(U \approx U_{\text{thr}}\), normalize \(U_{\text{thr}} = 1\), and the spike response can be obtained by scaling the membrane by 1: \(S = U \times 1 \Rightarrow \partial S^\sim/\partial U = 1\), where the tilde above \(S\) implies an approximation.

This is summarized as follows:

\[
\frac{\partial S^\sim}{\partial U} \leftarrow S = \begin{cases} 1, & \text{if } U > U_{\text{thr}} \\ 0, & \text{otherwise} \end{cases}
\]

where the left arrow denotes substitution.

The same neuron model described in (1) – (2) (a.k.a., snn.Leaky neuron from Tutorial 3) is implemented in PyTorch below. Don’t worry if you don’t understand this. This will be condensed into one line of code using snnTorch in a moment:

```python
# Leaky neuron model, overriding the backward pass with a custom function
class LeakySurrogate(nn.Module):
    def __init__(self, beta, threshold=1.0):
        super(LeakySurrogate, self).__init__()
```
# initialize decay rate beta and threshold
self.beta = beta
self.threshold = threshold
self.spike_op = self.SpikeOperator.apply

# the forward function is called each time we call Leaky
def forward(self, input_, mem):
    spk = self.spike_op((mem-self.threshold))  # call the Heaviside function
    reset = (spk * self.threshold).detach()  # removes spike_op gradient from reset
    mem = self.beta * mem + input_ - reset  # Eq (1)
    return spk, mem

# Forward pass: Heaviside function
# Backward pass: Override Dirac Delta with the Spike itself
@staticmethod
class SpikeOperator(torch.autograd.Function):
    @staticmethod
def forward(ctx, mem):
        spk = (mem > 0).float()  # Heaviside on the forward pass: Eq(2)
        ctx.save_for_backward(spk)  # store the spike for use in the backward pass
        return spk

    @staticmethod
def backward(ctx, grad_output):
        (spk,) = ctx.saved_tensors  # retrieve the spike
        grad = grad_output * spk  # scale the gradient by the spike: 1/0
        return grad

Note that the reset mechanism is detached from the computational graph, as the surrogate gradient should only be applied to $\partial S/\partial U$, and not $\partial R/\partial U$.

The above neuron is instantiated using:

```python
lif1 = LeakySurrogate(beta=0.9)
```

This neuron can be simulated using a for-loop, just as in previous tutorials, while PyTorch’s automatic differentiation (autodiff) mechanism keeps track of the gradient in the background.

The same thing can be accomplished by calling the `snn.Leaky` neuron. In fact, every time you call any neuron model from snnTorch, the `Spike Operator` surrogate gradient is applied to it by default:

```python
lif1 = snn.Leaky(beta=0.9)
```

If you would like to explore how this neuron behaves, then refer to Tutorial 3.
3. Backprop Through Time

Equation (4) only calculates the gradient for one single time step (referred to as the immediate influence in the figure below), but the backpropagation through time (BPTT) algorithm calculates the gradient from the loss to all descendants and sums them together.

The weight $W$ is applied at every time step, and so imagine a loss is also calculated at every time step. The influence of the weight on present and historical losses must be summed together to define the global gradient:

$$ \frac{\partial L}{\partial W} = \sum_t \frac{\partial L[t]}{\partial W} = \sum_t \sum_{s \leq t} \frac{\partial L[t]}{\partial W[s]} \frac{\partial W[s]}{\partial W} $$

The point of (5) is to ensure causality: by constraining $s \leq t$, we only account for the contribution of immediate and prior influences of $W$ on the loss. A recurrent system constrains the weight to be shared across all steps: $W[0] = W[1] = \ldots = W$. Therefore, a change in $W[s]$ will have the same effect on all $W$, which implies that $\frac{\partial W[s]}{\partial W} = 1$:

$$ \frac{\partial L}{\partial W} = \sum_t \sum_{s \leq t} \frac{\partial L[t]}{\partial W[s]} $$

As an example, isolate the prior influence due to $s = t - 1$ only; this means the backward pass must track back in time by one step. The influence of $W[t-1]$ on the loss can be written as:

$$ \frac{\partial L[t]}{\partial W[t-1]} = \frac{\partial L[t]}{\partial S[t]} \frac{\partial S[t]}{\partial U[t]} \frac{\partial U[t]}{\partial U[t-1]} \frac{\partial U[t-1]}{\partial I[t-1]} \frac{\partial I[t-1]}{\partial W[t-1]} $$

We have already dealt with all of these terms from (4), except for $\frac{\partial U[t]}{\partial U[t-1]}$. From (1), this temporal derivative term simply evaluates to $\beta$. So if we really wanted to, we now know enough to painstakingly calculate the derivative of every weight at every time step by hand, and it’d look something like this for a single neuron:
But thankfully, PyTorch’s autodiff takes care of that in the background for us.

4. Setting up the Loss / Output Decoding

In a conventional, non-spiking neural network, a supervised, multi-class classification problem takes the neuron with the highest activation and treats that as the predicted class.

In a spiking neural net, there are several options to interpreting the output spikes. The most common approaches are:

- **Rate coding**: Take the neuron with the highest firing rate (or spike count) as the predicted class
- **Latency coding**: Take the neuron that fires first as the predicted class

This might feel familiar to Tutorial 1 on neural encoding. The difference is that, here, we are interpreting (decoding) the output spikes, rather than encoding/converting raw input data into spikes.

Let’s focus on a rate code. When input data is passed to the network, we want the correct neuron class to emit the
most spikes over the course of the simulation run. This corresponds to the highest average firing frequency. One way
to achieve this is to increase the membrane potential of the correct class to $U > U_{\text{thr}}$, and that of incorrect classes to
$U < U_{\text{thr}}$. Applying the target to $U$ serves as a proxy for modulating spiking behavior from $S$.

This can be implemented by taking the softmax of the membrane potential for output neurons, where $C$ is the number
of output classes:

$$p_i[t] = \frac{e^{U_i[t]}}{\sum_{i=0}^{C} e^{U_i[t]}}$$

The cross-entropy between $p_i$ and the target $y_i \in \{0, 1\}^C$, which is a one-hot target vector, is obtained using:

$$\mathcal{L}_{CE}[t] = \sum_{i=0}^{C} y_i \log(p_i[t])$$

The practical effect is that the membrane potential of the correct class is encouraged to increase while those of incorrect
classes are reduced. In effect, this means the correct class is encouraged to fire at all time steps, while incorrect classes
are suppressed at all steps. This may not be the most efficient implementation of an SNN, but it is among the simplest.

This target is applied at every time step of the simulation, thus also generating a loss at every step. These losses are
then summed together at the end of the simulation:

$$\mathcal{L}_{CE} = \sum_t \mathcal{L}_{CE}[t]$$

This is just one of many possible ways to apply a loss function to a spiking neural network. A variety of approaches
are available to use in snnTorch (in the module `snn.functional`), and will be the subject of a future tutorial.

With all of the background theory having been taken care of, let’s finally dive into training a fully-connected spiking
neural net.

### 5. Setting up the Static MNIST Dataset

```python
# dataloader arguments
batch_size = 128
data_path='~/data/mnist'
dtype = torch.float
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")

# Define a transform
transform = transforms.Compose([  
    transforms.Resize((28, 28)),  
    transforms.Grayscale(),  
    transforms.ToTensor(),  
    transforms.Normalize((0,), (1,))])

mnist_train = datasets.MNIST(data_path, train=True, download=True, transform=transform)
mnist_test = datasets.MNIST(data_path, train=False, download=True, transform=transform)

# Create DataLoaders
train_loader = DataLoader(mnist_train, batch_size=batch_size, shuffle=True, drop_last=True)
test_loader = DataLoader(mnist_test, batch_size=batch_size, shuffle=True, drop_last=True)
```
6. Define the Network

```python
# Network Architecture
num_inputs = 28*28
num_hidden = 1000
num_outputs = 10

# Temporal Dynamics
num_steps = 25
beta = 0.95

class Net(nn.Module):
    def __init__(self):
        super().__init__()

        # Initialize layers
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.Leaky(beta=beta)
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        self.lif2 = snn.Leaky(beta=beta)

    def forward(self, x):
        # Initialize hidden states at t=0
        mem1 = self.lif1.init_leaky()
        mem2 = self.lif2.init_leaky()

        # Record the final layer
        spk2_rec = []
        mem2_rec = []

        for step in range(num_steps):
            cur1 = self.fc1(x)
            spk1, mem1 = self.lif1(cur1, mem1)
            cur2 = self.fc2(spk1)
            spk2, mem2 = self.lif2(cur2, mem2)
            spk2_rec.append(spk2)
            mem2_rec.append(mem2)

        return torch.stack(spk2_rec, dim=0), torch.stack(mem2_rec, dim=0)

# Load the network onto CUDA if available
net = Net().to(device)
```

The code in the `forward()` function will only be called once the input argument `x` is explicitly passed into `net`.

- `fc1` applies a linear transformation to all input pixels from the MNIST dataset;
- `lif1` integrates the weighted input over time, emitting a spike if the threshold condition is met;
- `fc2` applies a linear transformation to the output spikes of `lif1`;
- `lif2` is another spiking neuron layer, integrating the weighted spikes over time.
7. Training the SNN

7.1 Accuracy Metric

Below is a function that takes a batch of data, counts up all the spikes from each neuron (i.e., a rate code over the simulation time), and compares the index of the highest count with the actual target. If they match, then the network correctly predicted the target.

```python
def print_batch_accuracy(data, targets, train=False):
    output, _ = net(data.view(batch_size, -1))
    _, idx = output.sum(dim=0).max(1)
    acc = np.mean((targets == idx).detach().cpu().numpy())

    if train:
        print(f"Train set accuracy for a single minibatch: {acc*100:.2f}%")
    else:
        print(f"Test set accuracy for a single minibatch: {acc*100:.2f}%")
```

```python
def train_printer()
    print(f"Epoch {epoch}, Iteration {iter_counter}")
    print(f"Train Set Loss: {loss_hist[counter]:.2f}"
    print(f"Test Set Loss: {test_loss_hist[counter]:.2f}"
    print_batch_accuracy(data, targets, train=True)
    print_batch_accuracy(test_data, test_targets, train=False)
    print("\n")
```

---

7.2 Loss Definition

The `nn.CrossEntropyLoss` function in PyTorch automatically handles taking the softmax of the output layer as well as generating a loss at the output.

```python
loss = nn.CrossEntropyLoss()
```

7.3 Optimizer

Adam is a robust optimizer that performs well on recurrent networks, so let’s use that with a learning rate of $5 \times 10^{-4}$. 

---
7.4 One Iteration of Training

Take the first batch of data and load it onto CUDA if available.

```python
data, targets = next(iter(train_loader))
data = data.to(device)
targets = targets.to(device)
```

Flatten the input data to a vector of size 784 and pass it into the network.

```python
spk_rec, mem_rec = net(data.view(batch_size, -1))
```

```python
>>> print(mem_rec.size())
torch.Size([25, 128, 10])
```

The recording of the membrane potential is taken across:

- 25 time steps
- 128 samples of data
- 10 output neurons

We wish to calculate the loss at every time step, and sum these up together, as per Equation (10):

```python
# initialize the total loss value
loss_val = torch.zeros((1), dtype=dtype, device=device)

# sum loss at every step
for step in range(num_steps):
    loss_val += loss(mem_rec[step], targets)
```

```python
>>> print(f"Training loss: {loss_val.item():.3f}")
Training loss: 60.488
```

The loss is quite large, because it is summed over 25 time steps. The accuracy is also bad (it should be roughly around 10%) as the network is untrained:

```python
>>> print_batch_accuracy(data, targets, train=True)
Train set accuracy for a single minibatch: 10.16%
```

A single weight update is applied to the network as follows:

```python
# clear previously stored gradients
optimizer.zero_grad()

# calculate the gradients
loss_val.backward()

# weight update
optimizer.step()
```
Now, re-run the loss calculation and accuracy after a single iteration:

```python
# calculate new network outputs using the same data
spk_rec, mem_rec = net(data.view(batch_size, -1))

# initialize the total loss value
loss_val = torch.zeros((1), dtype=dtype, device=device)

# sum loss at every step
for step in range(num_steps):
    loss_val += loss(mem_rec[step], targets)

>>> print(f"Training loss: {loss_val.item():.3f}")
>>> print_batch_accuracy(data, targets, train=True)
Training loss: 47.384
Train set accuracy for a single minibatch: 33.59%
```

After only one iteration, the loss should have decreased and accuracy should have increased. Note how membrane potential is used to calculate the cross entropy loss, and spike count is used for the measure of accuracy. It is also possible to use the spike count in the loss (see Tutorial 6)

### 7.5 Training Loop

Let's combine everything into a training loop. We will train for one epoch (though feel free to increase num_epochs), exposing our network to each sample of data once.

```python
num_epochs = 1
loss_hist = []
test_loss_hist = []
counter = 0

# Outer training loop
for epoch in range(num_epochs):
    iter_counter = 0
    train_batch = iter(train_loader)

    # Minibatch training loop
    for data, targets in train_batch:
        data = data.to(device)
        targets = targets.to(device)

        # forward pass
        net.train()
        spk_rec, mem_rec = net(data.view(batch_size, -1))

        # initialize the loss & sum over time
        loss_val = torch.zeros((1), dtype=dtype, device=device)
        for step in range(num_steps):
            loss_val += loss(mem_rec[step], targets)

        # Gradient calculation + weight update
        optimizer.zero_grad()
```

(continues on next page)
The terminal will iteratively print out something like this every 50 iterations:

```
Epoch 0, Iteration 50
Train Set Loss: 12.63
Test Set Loss: 13.44
Train set accuracy for a single minibatch: 92.97%
Test set accuracy for a single minibatch: 90.62%
```

### 8. Results

#### 8.1 Plot Training/Test Loss

```
# Plot Loss
fig = plt.figure(facecolor="w", figsize=(10, 5))
plt.plot(loss_hist)
plt.plot(test_loss_hist)
plt.title("Loss Curves")
plt.legend(["Train Loss", "Test Loss"])
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.show()
```
The loss curves are noisy because the losses are tracked at every iteration, rather than averaging across multiple iterations.

### 8.2 Test Set Accuracy

This function iterates over all minibatches to obtain a measure of accuracy over the full 10,000 samples in the test set.

```python
total = 0
correct = 0

# drop_last switched to False to keep all samples
test_loader = DataLoader(mnist_test, batch_size=batch_size, shuffle=True, drop_last=False)

with torch.no_grad():
    net.eval()
    for data, targets in test_loader:
        data = data.to(device)
        targets = targets.to(device)

        # forward pass
        test_spk, _ = net(data.view(data.size(0), -1))

        # calculate total accuracy
        _, predicted = test_spk.sum(dim=0).max(1)
        total += targets.size(0)
        correct += (predicted == targets).sum().item()

>>> print(f"Total correctly classified test set images: {correct}/{total}")
>>> print(f"Test Set Accuracy: {100 * correct / total:.2f}%")
Total correctly classified test set images: 9387/10000
Test Set Accuracy: 93.87%
```
Voila! That’s it for static MNIST. Feel free to tweak the network parameters, hyperparameters, decay rate, using a learning rate scheduler etc. to see if you can improve the network performance.

Conclusion

Now you know how to construct and train a fully-connected network on a static dataset. The spiking neurons can also be adapted to other layer types, including convolutions and skip connections. Armed with this knowledge, you should now be able to build many different types of SNNs. In the next tutorial, you will learn how to train a spiking convolutional network, and simplify the amount of code required using the snn.backprop module.

Also, a special thanks to Bugra Kaytanli for providing valuable feedback on the tutorial.

If you like this project, please consider starring the repo on GitHub as it is the easiest and best way to support it.

Additional Resources

- Check out the snnTorch GitHub project here.

Tutorial 6 - Surrogate Gradient Descent in a Convolutional SNN

Tutorial written by Jason K. Eshraghian (www.ncg.ucsc.edu)

The snnTorch tutorial series is based on the following paper. If you find these resources or code useful in your work, please consider citing the following source:


Note:

This tutorial is a static non-editable version. Interactive, editable versions are available via the following links:

- Google Colab
- Local Notebook (download via GitHub)

Introduction

In this tutorial, you will:

- Learn how to use surrogate gradient descent to overcome the dead neuron problem
- Construct and train a convolutional spiking neural network
- Use a sequential container, nn.Sequential to simplify model construction
- Use the snn.backprop module to reduce the time it takes to design a neural network

Part of this tutorial was inspired by Friedemann Zenke’s extensive work on SNNs. Check out his repo on surrogate gradients here, and a favourite paper of mine: E. O. Neftci, H. Mostafa, F. Zenke, Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-based optimization to spiking neural networks. IEEE Signal Processing Magazine 36, 51–63.
At the end of the tutorial, we will train a convolutional spiking neural network (CSNN) using the MNIST dataset to perform image classification. The background theory follows on from Tutorials 2, 4 and 5, so feel free to go back if you need to brush up.

Install the latest PyPi distribution of snnTorch:

```
$ pip install snntorch
```

```python
# imports
import snntorch as snn
from snntorch import surrogate
from snntorch import backprop
from snntorch import functional as SF
from snntorch import utils
from snntorch import spikeplot as splt

import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np
import itertools
```

1. Surrogate Gradient Descent

Tutorial 5 raised the dead neuron problem. This arises because of the non-differentiability of spikes:

\[ S[t] = \Theta(U[t] - U_{\text{thr}}) \]

\[ \frac{\partial S}{\partial U} = \delta(U - U_{\text{thr}}) \in \{0, \infty\} \]

where \( \Theta(\cdot) \) is the Heaviside step function, and \( \delta(\cdot) \) is the Dirac-Delta function. We previously overcame this using the Spike-Operator approach, by assigning the spike to the derivative term: \( \partial \tilde{S} / \partial U \leftarrow S \in \{0, 1\} \). Another approach is to smooth the Heaviside function during the backward pass, which correspondingly smooths out the gradient of the Heaviside function.

Common smoothing functions include the sigmoid function, or the fast sigmoid function. The sigmoidal functions must also be shifted such that they are centered at the threshold \( U_{\text{thr}} \). Defining the overdrive of the membrane potential as \( U_{OD} = U - U_{\text{thr}} \):

\[ \tilde{S} = \frac{U_{OD}}{1 + k|U_{OD}|} \]

\[ \frac{\partial \tilde{S}}{\partial U} = \frac{1}{(k|U_{OD}| + 1)^2} \]

where \( k \) modulates how smooth the surrogate function is, and is treated as a hyperparameter. As \( k \) increases, the approximation converges towards the original derivative in (2):

\[ \frac{\partial \tilde{S}}{\partial U} \bigg|_{k \to \infty} = \delta(U - U_{\text{thr}}) \]
To summarize:

- **Forward Pass**
  - Determine $S$ using the shifted Heaviside function in (1)
  - Store $U$ for later use during the backward pass

- **Backward Pass**
  - Pass $U$ into (4) to calculate the derivative term

In the same way the Spike Operator approach was used in Tutorial 5, the gradient of the fast sigmoid function can override the Dirac-Delta function in a Leaky Integrate-and-Fire (LIF) neuron model:

```python
# Leaky neuron model, overriding the backward pass with a custom function
class LeakySigmoidSurrogate(nn.Module):
    def __init__(self, beta, threshold=1.0, k=25):
        super(Leaky_Surrogate, self).__init__():
        # initialize decay rate beta and threshold
        self.beta = beta
        self.threshold = threshold
        self.surrogate_func = self.FastSigmoid.apply

    # the forward function is called each time we call Leaky
    def forward(self, input_, mem):
        spk = self.surrogate_func((mem-self.threshold))  # call the Heaviside function
        reset = (spk - self.threshold).detach()
        mem = self.beta * mem + input_ - reset
        return spk, mem

# Forward pass: Heaviside function
# Backward pass: Override Dirac Delta with gradient of fast sigmoid
@staticmethod
```

(continues on next page)
class FastSigmoid(torch.autograd.Function):
    @staticmethod
    def forward(ctx, mem, k=25):
        ctx.save_for_backward(mem)  # store the membrane potential for use in the
        # forward pass
        ctx.k = k
        out = (mem > 0).float()  # Heaviside on the forward pass: Eq(1)
        return out

    @staticmethod
    def backward(ctx, grad_output):
        (mem,) = ctx.saved_tensors  # retrieve membrane potential
        grad_input = grad_output.clone()
        grad = grad_input / (ctx.k * torch.abs(mem) + 1.0) ** 2  # gradient of fast
        # sigmoid on backward pass: Eq(4)
        return grad, None

Better yet, all of that can be condensed by using the built-in module snn.surrogate from snnTorch, where \( k \) from (4) is denoted \( \text{slope} \). The surrogate gradient is passed into spike_grad as an argument:

spike_grad = surrogate.fast_sigmoid(slope=25)
beta = 0.5
lif1 = snn.Leaky(beta=beta, spike_grad=spike_grad)

To explore the other surrogate gradient functions available, take a look at the documentation here.

## 2. Setting up the CSNN

### 2.1 DataLoaders

Note that utils.data_subset() is called to reduce the size of the dataset by a factor of 10 to speed up training.

```python
# dataloader arguments
batch_size = 128
data_path='/data/mnist'
subset=10
dtype = torch.float
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")

# Define a transform
transform = transforms.Compose([  
    transforms.Resize((28, 28)),  
    transforms.Grayscale(),  
    transforms.ToTensor(),  
    transforms.Normalize((0,), (1,))] )

mnist_train = datasets.MNIST(data_path, train=True, download=True, transform=transform)
mnist_test = datasets.MNIST(data_path, train=False, download=True, transform=transform)
```
# reduce datasets by 10x to speed up training
utils.data_subset(mnist_train, subset)
utils.data_subset(mnist_test, subset)

# Create DataLoaders
train_loader = DataLoader(mnist_train, batch_size=batch_size, shuffle=True, drop_last=True)
test_loader = DataLoader(mnist_test, batch_size=batch_size, shuffle=True, drop_last=True)

## 2.2 Define the Network

The convolutional network architecture to be used is: 12C5-MP2-64C5-MP2-1024FC10

- 12C5 is a 5 x 5 convolutional kernel with 12 filters
- MP2 is a 2 x 2 max-pooling function
- 1024FC10 is a fully-connected layer that maps 1,024 neurons to 10 outputs

# neuron and simulation parameters
spike_grad = surrogate.fast_sigmoid(slope=25)
beta = 0.5
num_steps = 50

# Define Network
class Net(nn.Module):
    def __init__(self):
        super().__init__()

        # Initialize layers
        self.conv1 = nn.Conv2d(1, 12, 5)
        self.lif1 = snn.Leaky(beta=beta, spike_grad=spike_grad)
        self.conv2 = nn.Conv2d(12, 64, 5)
        self.lif2 = snn.Leaky(beta=beta, spike_grad=spike_grad)
        self.fc1 = nn.Linear(64*4*4, 10)
        self.lif3 = snn.Leaky(beta=beta, spike_grad=spike_grad)

    def forward(self, x):

        # Initialize hidden states and outputs at t=0
        mem1 = self.lif1.init_leaky()
        mem2 = self.lif2.init_leaky()
        mem3 = self.lif3.init_leaky()

        # Record the final layer
        spk3_rec = []
        mem3_rec = []

        for step in range(num_steps):
            cur1 = F.max_pool2d(self.conv1(x), 2)
            spk1, mem1 = self.lif1(cur1, mem1)
            cur2 = F.max_pool2d(self.conv2(spk1), 2)

            spk2, mem2 = self.lif2(cur2, mem2)
            spk3, mem3 = self.lif3(spk2, mem3)
            spk3_rec.append(spk3)
spk2, mem2 = self.lif2(cur2, mem2)
cur3 = self.fc1(spk2.view(batch_size, -1))
spk3, mem3 = self.lif3(cur3, mem3)

spk3_rec.append(spk3)
mem3_rec.append(mem3)

return torch.stack(spk3_rec), torch.stack(mem3_rec)

In the previous tutorial, the network was wrapped inside of a class, as shown above. With increasing network complexity, this adds a lot of boilerplate code that we might wish to avoid. Alternatively, the nn.Sequential method can be used instead:

```python
# Initialize Network
net = nn.Sequential(nn.Conv2d(1, 12, 5),
                     nn.MaxPool2d(2),
                     snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True),
                     nn.Conv2d(12, 64, 5),
                     nn.MaxPool2d(2),
                     snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True),
                     nn.Flatten(),
                     nn.Linear(64*4*4, 10),
                     snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True, output=True))
```

The init_hidden argument initializes the hidden states of the neuron (here, membrane potential). This takes place in the background as an instance variable. If init_hidden is activated, the membrane potential is not explicitly returned to the user, ensuring only the output spikes are sequentially passed through the layers wrapped in nn.Sequential.

To train a model using the final layer’s membrane potential, set the argument output=True. This enables the final layer to return both the spike and membrane potential response of the neuron.

### 2.3 Forward-Pass

A forward pass across a simulation duration of num_steps looks like this:

```python
data, targets = next(iter(train_loader))
data = data.to(device)
targets = targets.to(device)

for step in range(num_steps):
    spk_out, mem_out = net(data)
```

Wrap that in a function, recording the membrane potential and spike response over time:

```python
def forward_pass(net, num_steps, data):
    mem_rec = []
    spk_rec = []
    utils.reset(net)  # resets hidden states for all LIF neurons in net

    for step in range(num_steps):
        ```
spk_out, mem_out = net(data)
spk_rec.append(spk_out)
mem_rec.append(mem_out)

return torch.stack(spk_rec), torch.stack(mem_rec)

spk_rec, mem_rec = forward_pass(net, num_steps, data)

## 3. Training Loop

### 3.1 Loss Using snn.Functional

In the previous tutorial, the Cross Entropy Loss between the membrane potential of the output neurons and the target was used to train the network. This time, the total number of spikes from each neuron will be used to calculate the Cross Entropy instead.

A variety of loss functions are included in the `snn.functional` module, which is analogous to `torch.nn.functional` in PyTorch. These implement a mix of cross entropy and mean square error losses, are applied to spikes and/or membrane potential, to train a rate or latency-coded network.

The approach below applies the cross entropy loss to the output spike count in order train a rate-coded network:

```python
# already imported snntorch.functional as SF
loss_fn = SF.ce_rate_loss()

The recordings of the spike are passed as the first argument to `loss_fn`, and the target neuron index as the second argument to generate a loss. The documentation provides further information and examples.

loss_val = loss_fn(spk_rec, targets)

>>> print(f"The loss from an untrained network is {loss_val.item():.3f}"")
The loss from an untrained network is 2.303

### 3.2 Accuracy Using snn.Functional

The `SF.accuracy_rate()` function works similarly, in that the predicted output spikes and actual targets are supplied as arguments. `accuracy_rate` assumes a rate code is used to interpret the output by checking if the index of the neuron with the highest spike count matches the target index.

```python
acc = SF.accuracy_rate(spk_rec, targets)

>>> print(f"The accuracy of a single batch using an untrained network is {acc*100:.3f}%")
The accuracy of a single batch using an untrained network is 10.938%
```

As the above function only returns the accuracy of a single batch of data, the following function returns the accuracy on the entire DataLoader object:

```python
def batch_accuracy(train_loader, net, num_steps):
    with torch.no_grad():
```
total = 0
acc = 0

net.eval()

train_loader = iter(train_loader)
for data, targets in train_loader:
data = data.to(device)
targets = targets.to(device)
spk_rec, _ = forward_pass(net, num_steps, data)

acc += SF.accuracy_rate(spk_rec, targets) * spk_rec.size(1)
total += spk_rec.size(1)

return acc/total

test_acc = batch_accuracy(test_loader, net, num_steps)

>>> print(f"The total accuracy on the test set is: {test_acc * 100:.2f}%")
The total accuracy on the test set is: 8.59%

3.3 Training Automation Using snn.backprop

Training SNNs can become arduous even with simple networks, so the snn.backprop module is here to reduce some of this effort.

The backprop.BPTT function automatically performs a single epoch of training, where you need only provide the training parameters, dataloader, and several other arguments. The average loss across iterations is returned. The argument time_var indicates whether the input data is time-varying. As we are using the MNIST dataset, we explicitly specify time_var=False.

The following code block may take a while to run. If you are not connected to GPU, then consider reducing num_epochs.

optimizer = torch.optim.Adam(net.parameters(), lr=1e-2, betas=(0.9, 0.999))
num_epochs = 10
test_acc_hist = []

# training loop
for epoch in range(num_epochs):
    avg_loss = backprop.BPTT(net, train_loader, optimizer=optimizer, criterion=loss_fn,
                              num_steps=num_steps, time_var=False, device=device)

    print(f"Epoch {epoch}, Train Loss: {avg_loss.item():.2f}"")

    # Test set accuracy
    test_acc = batch_accuracy(test_loader, net, num_steps)
    test_acc_hist.append(test_acc)

    print(f"Epoch {epoch}, Test Acc: {test_acc * 100:.2f}%\n")

The output should look something like this:
Despite having selected some fairly generic values and architectures, the test set accuracy should be fairly competitive given the brief training run!

4. Results

4.1 Plot Test Accuracy

```python
# Plot Loss
fig = plt.figure(facecolor="w")
plt.plot(test_acc_hist)
plt.title("Test Set Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.show()
```

Test Set Accuracy

[Graph showing test set accuracy over epochs]
4.2 Spike Counter

Run a forward pass on a batch of data to obtain spike and membrane readings.

```python
spk_rec, mem_rec = forward_pass(net, num_steps, data)
```

Changing `idx` allows you to index into various samples from the simulated minibatch. Use `splt.spike_count` to explore the spiking behaviour of a few different samples!

Note: if you are running the notebook locally on your desktop, please uncomment the line below and modify the path to your ffmpeg.exe

```python
from IPython.display import HTML

idx = 0

fig, ax = plt.subplots(facecolor='w', figsize=(12, 7))
labels=['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

# plt.rcParams['animation.ffmpeg_path'] = 'C:\\path\\to\\your\\ffmpeg.exe'

# Plot spike count histogram
anim = splt.spike_count(spk_rec[:, idx].detach().cpu(), fig, ax, labels=labels, animate=True, interpolate=4)

HTML(anim.to_html5_video())
# anim.save("spike_bar.mp4")
```

```python
>>> print(f"The target label is: {targets[idx]}")
The target label is: 3
```

Conclusion

You should now have a grasp of the basic features of snnTorch and be able to start running your own experiments. In the next tutorial, we will train a network using a neuromorphic dataset.

If you like this project, please consider starring the repo on GitHub as it is the easiest and best way to support it.
Additional Resources

- Check out the snnTorch GitHub project here.

Tutorial 7 - Neuromorphic Datasets with Tonic + snnTorch

Tutorial written by Gregor Lenz (https://lenzgregor.com) and Jason K. Eshraghian (www.ncg.ucsc.edu)

The snnTorch tutorial series is based on the following paper. If you find these resources or code useful in your work, please consider citing the following source:


Note:

This tutorial is a static non-editable version. Interactive, editable versions are available via the following links:

- Google Colab
- Local Notebook (download via GitHub)

Introduction

In this tutorial, you will:

- Learn how to load neuromorphic datasets using Tonic
- Make use of caching to speed up dataloading
- Train a CSNN with the Neuromorphic-MNIST Dataset

Install the latest PyPi distribution of snnTorch:

```
pip install tonic
pip install snntorch
```

1. Using Tonic to Load Neuromorphic Datasets

Loading datasets from neuromorphic sensors is made super simple thanks to Tonic, which works much like PyTorch vision.

Let’s start by loading the neuromorphic version of the MNIST dataset, called N-MNIST. We can have a look at some raw events to get a feel for what we’re working with.

```
import tonic

dataset = tonic.datasets.NMNIST(save_to='./data', train=True)

events, target = dataset[0]
```
Each row corresponds to a single event, which consists of four parameters: (x-coordinate, y-coordinate, timestamp, polarity).

- x & y co-ordinates correspond to an address in a 34 × 34 grid.
- The timestamp of the event is recorded in microseconds.
- The polarity refers to whether an on-spike (+1) or an off-spike (-1) occurred; i.e., an increase in brightness or a decrease in brightness.

If we were to accumulate those events over time and plot the bins as images, it looks like this:

```python
>>> tonic.utils.plot_event_grid(events)
```

![Event Grid](image)

### 1.1 Transformations

However, neural nets don’t take lists of events as input. The raw data must be converted into a suitable representation, such as a tensor. We can choose a set of transforms to apply to our data before feeding it to our network. The neuromorphic camera sensor has a temporal resolution of microseconds, which when converted into a dense representation, ends up as a very large tensor. That is why we bin events into a smaller number of frames using the `ToFrame` transformation, which reduces temporal precision but also allows us to work with it in a dense format.

- `time_window=1000` integrates events into 1000 μs bins
- Denoise removes isolated, one-off events. If no event occurs within a neighbourhood of 1 pixel across `filter_time` microseconds, the event is filtered. Smaller `filter_time` will filter more events.

```python
import tonic.transforms as transforms

sensor_size = tonic.datasets.NMNIST.sensor_size

# Denoise removes isolated, one-off events
# time_window
frame_transform = transforms.Compose([transforms.Denoise(filter_time=10000),
                                      transforms.ToFrame(sensor_size=sensor_size,
                                      time_window=1000)])

trainset = tonic.datasets.NMNIST(save_to='./data', transform=frame_transform, train=True)

testset = tonic.datasets.NMNIST(save_to='./data', transform=frame_transform, train=False)
```
1.2 Fast DataLoading

The original data is stored in a format that is slow to read. To speed up dataloading, we can make use of disk caching and batching. That means that once files are loaded from the original dataset, they are written to the disk.

Because event recordings have different lengths, we are going to provide a collation function `tonic.collation.PadTensors()` that will pad out shorter recordings to ensure all samples in a batch have the same dimensions.

```python
from torch.utils.data import DataLoader
from tonic import DiskCachedDataset

cached_trainset = DiskCachedDataset(trainset, cache_path='./cache/nmnist/train')
cached_dataloader = DataLoader(cached_trainset)

batch_size = 128
trainloader = DataLoader(cached_trainset, batch_size=batch_size, collate_fn=tonic.collation.PadTensors())

def load_sample_batched():
    events, target = next(iter(cached_dataloader))

>>> %timeit -o -r 10 load_sample_batched()
4.2 ms ± 1.19 µs per loop (mean ± std. dev. of 10 runs, 100 loops each)
```

By using disk caching and a PyTorch dataloader with multithreading and batching support, we have significantly reduced loading times.

If you have a large amount of RAM available, you can speed up dataloading further by caching to main memory instead of to disk:

```python
from tonic import MemoryCachedDataset

cached_trainset = MemoryCachedDataset(trainset)
```

2. Training our network using frames created from events

Now let’s actually train a network on the N-MNIST classification task. We start by defining our caching wrappers anddataloaders. While doing that, we’re also going to apply some augmentations to the training data. The samples we receive from the cached dataset are frames, so we can make use of PyTorch Vision to apply whatever random transform we would like.

```python
import torch
import torchvision

transform = tonic.transforms.Compose(
    [torch.from_numpy,
     torchvision.transforms.RandomRotation([-10, 10])])

cached_trainset = DiskCachedDataset(trainset, transform=transform, cache_path='./cache/nmnist/train')

# no augmentations for the testset
cached_testset = DiskCachedDataset(testset, cache_path='./cache/nmnist/test')
```

(continues on next page)
A mini-batch now has the dimensions (time steps, batch size, channels, height, width). The number of time steps will be set to that of the longest recording in the mini-batch, and all other samples will be padded with zeros to match it.

```python
>>> event_tensor, target = next(iter(trainloader))
>>> print(event_tensor.shape)
torch.Size([311, 128, 2, 34, 34])
```

### 2.1 Defining our network

We will use snnTorch + PyTorch to construct a CSNN, just as in the previous tutorial. The convolutional network architecture to be used is: 12C5-MP2-32C5-MP2-800FC10

- 12C5 is a $5 \times 5$ convolutional kernel with 12 filters
- MP2 is a $2 \times 2$ max-pooling function
- 800FC10 is a fully-connected layer that maps 800 neurons to 10 outputs

```python
import snntorch as snn
from snntorch import surrogate
from snntorch import functional as SF
from snntorch import spikeplot as splt
from snntorch import utils
import torch.nn as nn

device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")

# neuron and simulation parameters
spike_grad = surrogate.atan()
beta = 0.5

# Initialize Network
net = nn.Sequential(nn.Conv2d(2, 12, 5),
                    nn.MaxPool2d(2),
                    snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True),
                    nn.Conv2d(12, 32, 5),
                    nn.MaxPool2d(2),
                    snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True),
                    nn.Flatten(),
                    nn.Linear(32*5*5, 10),
                    snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True, output=True)).to(device)
```
# this time, we won't return membrane as we don't need it

def forward_pass(net, data):
    spk_rec = []
    utils.reset(net)  # resets hidden states for all LIF neurons in net

    for step in range(data.size(0)):  # data.size(0) = number of time steps
        spk_out, mem_out = net(data[step])
        spk_rec.append(spk_out)

    return torch.stack(spk_rec)

2.2 Training

In the previous tutorial, Cross Entropy Loss was applied to the total spike count to maximize the number of spikes from the correct class.

Another option from the snn.functional module is to specify the target number of spikes from correct and incorrect classes. The approach below uses the Mean Square Error Spike Count Loss, which aims to elicit spikes from the correct class 80% of the time, and 20% of the time from incorrect classes. Encouraging incorrect neurons to fire could be motivated to avoid dead neurons.

```python
optimizer = torch.optim.Adam(net.parameters(), lr=2e-2, betas=(0.9, 0.999))
loss_fn = SF.mse_count_loss(correct_rate=0.8, incorrect_rate=0.2)
```

Training neuromorphic data is expensive as it requires sequentially iterating through many time steps (approximately 300 time steps in the N-MNIST dataset). The following simulation will take some time, so we will just stick to training across 50 iterations (which is roughly 1/10th of a full epoch). Feel free to change num_iters if you have more time to kill. As we are printing results at each iteration, the results will be quite noisy and will also take some time before we start to see any sort of improvement.

In our own experiments, it took about 20 iterations before we saw any improvement, and after 50 iterations, managed to crack ~60% accuracy.

Warning: the following simulation will take a while. Go make yourself a coffee, or ten.

```python
num_epochs = 1
num_iters = 50

loss_hist = []
acc_hist = []

# training loop
for epoch in range(num_epochs):
    for i, (data, targets) in enumerate(iter(trainloader)):
        data = data.to(device)
        targets = targets.to(device)

        net.train()
        spk_rec = forward_pass(net, data)
        loss_val = loss_fn(spk_rec, targets)

        # Gradient calculation + weight update
```

(continues on next page)
optimizer.zero_grad()
loss_val.backward()
optimizer.step()

# Store loss history for future plotting
loss_hist.append(loss_val.item())

print(f"Epoch {epoch}, Iteration {i} \nTrain Loss: {loss_val.item():.2f}\n")

acc = SF.accuracy_rate(spk_rec, targets)
acc_hist.append(acc)
print(f"Accuracy: {acc * 100:.2f}%\n")

# training loop breaks after 50 iterations
if i == num_iters:
    break

The output should look something like this:

Epoch 0, Iteration 0
Train Loss: 31.00
Accuracy: 10.16%

Epoch 0, Iteration 1
Train Loss: 30.58
Accuracy: 13.28%

And after some more time:

Epoch 0, Iteration 49
Train Loss: 8.78
Accuracy: 47.66%

Epoch 0, Iteration 50
Train Loss: 8.43
Accuracy: 56.25%

3. Results

3.1 Plot Test Accuracy

import matplotlib.pyplot as plt

# Plot Loss
fig = plt.figure(facecolor="w")
plt.plot(acc_hist)
plt.title("Train Set Accuracy")
plt.xlabel("Iteration")
plt.ylabel("Accuracy")
plt.show()
3.2 Spike Counter

Run a forward pass on a batch of data to obtain spike recordings.

```
spk_rec = forward_pass(net, data)
```

Changing `idx` allows you to index into various samples from the simulated minibatch. Use `splt.spike_count` to explore the spiking behaviour of a few different samples. Generating the following animation will take some time.

Note: if you are running the notebook locally on your desktop, please uncomment the line below and modify the path to your ffmpeg.exe

```
from IPython.display import HTML

idx = 0
fig, ax = plt.subplots(facecolor='w', figsize=(12, 7))
labels=['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
print(f"The target label is: {targets[idx]}")

# plt.rcParams['animation.ffmpeg_path'] = 'C:\\path\\to\\your\\ffmpeg.exe'

# Plot spike count histogram
anim = splt.spike_count(spk_rec[:, idx].detach().cpu(), fig, ax, labels=labels,
                        animate=True, interpolate=1)

HTML(anim.to_html5_video())
# anim.save("spike_bar.mp4")
```

The target label is: 3
Conclusion

If you made it this far, then congratulations - you have the patience of a monk. You should now also understand how to load neuromorphic datasets using Tonic and then train a network using snnTorch. In the next tutorial, we will learn more advanced techniques, such as introducing long-term temporal dynamics into our SNNs.

If you like this project, please consider starring the repo on GitHub as it is the easiest and best way to support it.

Additional Resources

- Check out the snnTorch GitHub project here.
- The Tonic GitHub project can be found here.
- For further information about how N-MNIST was created, please refer to Garrick Orchard’s website here.

Accelerating snnTorch on IPUs

Tutorial written by Jason K. Eshraghian and Vincent Sun

The snnTorch tutorial series is based on the following paper. If you find these resources or code useful in your work, please consider citing the following source:


Note:

This tutorial is a static non-editable version. An editable script is available via the following link:

- Python Script (download via GitHub)

Introduction

Spiking neural networks (SNNs) have achieved orders of magnitude improvement in terms of energy consumption and latency when performing inference with deep learning workloads. But in a twist of irony, using error backpropagation to train SNNs becomes more expensive than non-spiking network when trained on CPUs and GPUs. The additional temporal dimension must be accounted for, and memory complexity increases linearly with time when a network is trained using the backpropagation-through-time algorithm.

An alternative build of snnTorch has been optimized for Graphcore’s Intelligence Processing Units (IPUs). IPUs are custom accelerators tailored for deep learning workloads, and adopt multi-instruction multi-data (MIMD) parallelism by running individual processing threads on smaller blocks of data. This is an ideal fit for partitions of spiking neuron dynamical state equations that must be sequentially processed, and cannot be vectorized.

In this tutorial, you will:

- Learn how to train a SNN accelerated using IPUs.
Ensure up-to-date versions of poptorch and the Poplar SDK are installed. Refer to Graphcore's documentation for installation instructions.

Install snntorch-ipu in an environment that does not have snntorch pre-installed to avoid package conflicts:

```bash
!pip install snntorch-ipu
```

Import the required Python packages:

```python
import torch, torch.nn as nn
import popart, poptorch
import snntorch as snn
import snntorch.functional as SF
```

**DataLoading**

Load in the MNIST dataset.

```python
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

batch_size = 128
data_path='~/data/mnist'

# Define a transform
transform = transforms.Compose([transforms.Resize((28, 28)), transforms.Grayscale(), transforms.ToTensor(), transforms.Normalize((0,), (1,))])

mnist_train = datasets.MNIST(data_path, train=True, download=True, transform=transform)
mnist_test = datasets.MNIST(data_path, train=False, download=True, transform=transform)

# Train using full precision 32-flt
opts = poptorch.Options()
opts.Precision.halfFloatCasting(poptorch.HalfFloatCastingBehavior.HalfUpcastToFloat)

# Create DataLoaders
train_loader = poptorch.DataLoader(options=opts, dataset=mnist_train, batch_size=batch_size, shuffle=True, num_workers=20)
test_loader = poptorch.DataLoader(options=opts, dataset=mnist_test, batch_size=batch_size, shuffle=True, num_workers=20)
```
Define Network

Let’s simulate our network for 25 time steps using a slow state-decay rate for our spiking neurons:

```python
num_steps = 25
beta = 0.9
```

We will now construct a vanilla SNN model. When training on IPUs, note that the loss function must be wrapped within the model class. The full code will look this:

```python
class Model(torch.nn.Module):
    def __init__(self):
        super().__init__()
        num_inputs = 784
        num_hidden = 1000
        num_outputs = 10

        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.Leaky(beta=beta)
        self.fc2 = nn.Linear(num_hidden, num_output)
        self.lif2 = snn.Leaky(beta=beta)

        # Cross-Entropy Spike Count Loss
        self.loss_fn = SF.ce_count_loss()

    def forward(self, x, labels=None):
        mem1 = self.lif1.init_leaky()
        mem2 = self.lif2.init_leaky()

        spk2_rec = []
        mem2_rec = []

        for step in range(num_steps):
            cur1 = self.fc1(x.view(batch_size,-1))
            spk1, mem1 = self.lif1(cur1, mem1)
            cur2 = self.fc2(spk1)
            spk2, mem2 = self.lif2(cur2, mem2)

            spk2_rec.append(spk2)
            mem2_rec.append(mem2)

        spk2_rec = torch.stack(spk2_rec)
        mem2_rec = torch.stack(mem2_rec)

        if self.training:
            return spk2_rec, poptorch.identity_loss(self.loss_fn(mem2_rec, labels), "none")
        return spk2_rec
```

Let’s quickly break this down.

Constructing the model is the same as all previous tutorials. We apply spiking neuron nodes at the end of each dense layer:
By default, the surrogate gradient of the spiking neurons will be a straight through estimator. Fast Sigmoid and Sigmoid options are also available if you prefer to use those:

```python
from snntorch import surrogate

self.lif1 = snn.Leaky(beta=beta, spike_grad = surrogate.fast_sigmoid())
```

The loss function will count up the total number of spikes from each output neuron and apply the Cross Entropy Loss:

```python
self.loss_fn = SF.ce_count_loss()
```

Now we define the forward pass. Initialize the hidden state of each spiking neuron by calling the following functions:

```python
mem1 = self.lif1.init_leaky()
mem2 = self.lif2.init_leaky()
```

Next, run the for-loop to simulate the SNN over 25 time steps. The input data is flattened using `.view(batch_size, -1)` to make it compatible with a dense input layer.

```python
for step in range(num_steps):
    cur1 = self.fc1(x.view(batch_size,-1))
    spk1, mem1 = self.lif1(cur1, mem1)
    cur2 = self.fc2(spk1)
    spk2, mem2 = self.lif2(cur2, mem2)
```

The loss is applied using the function `poptorch.identity_loss(self.loss_fn(mem2_rec, labels), "none")`.

### Training on IPUs

Now, the full training loop is run across 10 epochs. Note the optimizer is called from `poptorch`. Otherwise, the training process is much the same as in typical use of snnTorch.

```python
net = Model()
op = poptorch.optim.Adam(net.parameters(), lr=0.001, betas=(0.9, 0.999))
poptorch_model = poptorch.trainingModel(net, options=opts, optimizer=optimizer)
epochs = 10
for epoch in tqdm(range(epochs), desc="epochs"):
    correct = 0.0
    for i, (data, labels) in enumerate(train_loader):
        output, loss = poptorch_model(data, labels)
        if i % 250 == 0:
            _, pred = output.sum(dim=0).max(1)
            correct = (labels == pred).sum().item()/len(labels)
```
# Accuracy on a single batch

```python
print("Accuracy: ", correct)
```

The model will first be compiled, after which, the training process will commence. The accuracy will be printed out for individual minibatches on the training set to keep this tutorial quick and minimal.

## Conclusion

Our initial benchmarks on show improvements of up to 10x improvements over CUDA accelerated SNNs in mixed-precision training throughput across a variety of neuron models. A detailed benchmark and blog highlighting additional features are currently under construction.

- For a detailed tutorial of spiking neurons, neural nets, encoding, and training using neuromorphic datasets, check out the **snnTorch tutorial series**.
- For more information on the features of snnTorch, check out the **documentation at this link**.
- If you have ideas, suggestions or would like to find ways to get involved, then check out the **snnTorch GitHub project here**.

## Population Coding in Spiking Neural Nets

Tutorial written by Jason K. Eshraghian ([www.jasoneshraghian.com](http://www.jasoneshraghian.com))

The snnTorch tutorial series is based on the following paper. If you find these resources or code useful in your work, please consider citing the following source:


### Note:

This tutorial is a static non-editable version. Interactive, editable versions are available via the following links:

- Google Colab
- Local Notebook (download via GitHub)

### Introduction

It is thought that rate codes alone cannot be the dominant encoding mechanism in the primary cortex. One of several reasons is because the average neuronal firing rate is roughly 0.1 – 1 Hz, which is far slower than the reaction response time of animals and humans.

But if we pool together multiple neurons and count their spikes together, then it becomes possible to measure a firing rate for a population of neurons in a very short window of time. Population coding adds some credibility to the plausibility of rate-encoding mechanisms.
In this tutorial, you will:

- Learn how to train a population coded network. Instead of assigning one neuron per class, we will extend this to multiple neurons per class, and aggregate their spikes together.

```bash
!pip install snntorch
```

```python
import torch, torch.nn as nn
import snntorch as snn
```

### DataLoading

Define variables for dataloading.

```python
batch_size = 128
data_path='data/fmnist'
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
```

Load FashionMNIST dataset.

```python
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Define a transform
transform = transforms.Compose([transforms.Resize((28, 28)), transforms.Grayscale(), transforms.ToTensor(), transforms.Normalize((0,), (1,))])

fmnist_train = datasets.FashionMNIST(data_path, train=True, download=True)
```

(continues on next page)
Define Network

Let’s compare the performance of a pair of networks both with and without population coding, and train them for one single time step.

```python
# network parameters
num_inputs = 28*28
num_hidden = 128
num_outputs = 10
num_steps = 1

# spiking neuron parameters
beta = 0.9  # neuron decay rate
grad = surrogate.fast_sigmoid()
```

Without population coding

Let’s just use a simple 2-layer dense spiking network.

```python
net = nn.Sequential(nn.Flatten(),
    nn.Linear(num_inputs, num_hidden),
    snn.Leaky(beta=beta, spike_grad=grad, init_hidden=True),
    nn.Linear(num_hidden, num_outputs),
    snn.Leaky(beta=beta, spike_grad=grad, init_hidden=True, output=True)).to(device)
```

With population coding

Instead of 10 output neurons corresponding to 10 output classes, we will use 500 output neurons. This means each output class has 50 neurons randomly assigned to it.

```python
pop_outputs = 500
net_pop = nn.Sequential(nn.Flatten(),
    nn.Linear(num_inputs, num_hidden),
    snn.Leaky(beta=beta, spike_grad=grad, init_hidden=True),
    nn.Linear(num_hidden, pop_outputs),
    snn.Leaky(beta=beta, spike_grad=grad, init_hidden=True, output=True))
```
Training

Without population coding

Define the optimizer and loss function. Here, we use the MSE Count Loss, which counts up the total number of output spikes at the end of the simulation run.

The correct class has a target firing probability of 100%, and incorrect classes are set to 0%.

```python
import snntorch.functional as SF
optimizer = torch.optim.Adam(net.parameters(), lr=2e-3, betas=(0.9, 0.999))
loss_fn = SF.mse_count_loss(correct_rate=1.0, incorrect_rate=0.0)
```

We will also define a simple test accuracy function that predicts the correct class based on the neuron with the highest spike count.

```python
from snntorch import utils
def test_accuracy(data_loader, net, num_steps, population_code=False, num_classes=False):
    with torch.no_grad():
        total = 0
        acc = 0
        net.eval()
        data_loader = iter(data_loader)
        for data, targets in data_loader:
            data = data.to(device)
            targets = targets.to(device)
            utils.reset(net)
            spk_rec, _ = net(data)
            if population_code:
                acc += SF.accuracy_rate(spk_rec.unsqueeze(0), targets, population_code=True, num_classes=10) * spk_rec.size(1)
            else:
                acc += SF.accuracy_rate(spk_rec.unsqueeze(0), targets) * spk_rec.size(1)
            total += spk_rec.size(1)
        return acc/total
```

Let's run the training loop. Note that we are only training for 1 time step. I.e., each neuron only has the opportunity to fire once. As a result, we might not expect the network to perform too well here.

```python
from snntorch import backprop
num_epochs = 5
```
# training loop
for epoch in range(num_epochs):
    
    avg_loss = backprop.BPTT(net, train_loader, num_steps=num_steps,
    optimizer=optimizer, criterion=loss_fn, time_var=False,
    device=device)

    print(f"Epoch: {epoch}\n")
    print(f"Test set accuracy: {test_accuracy(test_loader, net, num_steps)*100:.3f}%\n")

    >> Epoch: 0
    >> Test set accuracy: 59.421%

    >> Epoch: 1
    >> Test set accuracy: 61.889%

While there are ways to improve single time-step performance, e.g., by applying the loss to the membrane potential, one single time-step is extremely challenging to train a network competitively using rate codes.

With population coding

Let’s modify the loss function to specify that population coding should be enabled. We must also specify the number of classes. This means that there will be a total of 50 neurons per class = 500 neurons / 10 classes.

    loss_fn = SF.mse_count_loss(correct_rate=1.0, incorrect_rate=0.0, population_code=True,
    num_classes=10)
    optimizer = torch.optim.Adam(net_pop.parameters(), lr=2e-3, betas=(0.9, 0.999))

num_epochs = 5

# training loop
for epoch in range(num_epochs):
    
    avg_loss = backprop.BPTT(net_pop, train_loader, num_steps=num_steps,
    optimizer=optimizer, criterion=loss_fn, time_var=False,
    device=device)

    print(f"Epoch: {epoch}\n")
    print(f"Test set accuracy: {test_accuracy(test_loader, net_pop, num_steps,\n    population_code=True, num_classes=10)*100:.3f}%\n")

    >> Epoch: 0
    >> Test set accuracy: 80.501%

    >> Epoch: 1
    >> Test set accuracy: 82.690%

Even though we are only training on one time-step, introducing additional output neurons has immediately enabled better performance.
Conclusion

The performance boost from population coding may start to fade as the number of time steps increases. But it may also be preferable to increasing time steps as PyTorch is optimized for handling matrix-vector products, rather than sequential, step-by-step operations over time.

• For a detailed tutorial of spiking neurons, neural nets, encoding, and training using neuromorphic datasets, check out the snnTorch tutorial series.
• For more information on the features of snnTorch, check out the documentation at this link.
• If you have ideas, suggestions or would like to find ways to get involved, then check out the snnTorch GitHub project here.

1.11.14 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

If you are reporting a bug, please include:

• Your operating system name and version.
• Any details about your local setup that might be helpful in troubleshooting.
• Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.
Write Documentation

snntorch could always use more documentation, whether as part of the official snntorch docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/jeshraghian/snntorch/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here’s how to set up snntorch for local development.

1. Fork the snntorch repo on GitHub.
2. Clone your fork locally:

   $ git clone git@github.com:your_name_here/snntorch.git

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   $ mkvirtualenv snntorch
   $ cd snntorch/
   $ python setup.py develop

4. Create a branch for local development:

   $ git checkout -b name-of-your-bugfix-or-feature

   Now you can make your changes locally.

5. When you’re done making changes, check that your changes pass flake8 and the tests. In addition, ensure that your code is formatted using black:

   $ flake8 snntorch tests
   $ black snntorch tests
   $ python setup.py test or pytest

   To get flake8, black and tox, just pip install them into your virtualenv. If you wish, you can add pre-commit hooks for both flake8 and black to make all formatting easier.

6. Commit your changes and push your branch to GitHub:

   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature

7. Submit a pull request through the GitHub website.
Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.7, 3.8, 3.9 and for PyPy. Check https://github.com/jeshraghian/snntorch/actions and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ pytest tests.test_snntorch
```

Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version patch # possible: major / minor / patch
$ git push
$ git push --tags
```

GitHub Actions will then deploy to PyPI if tests pass.

1.11.15 History

0.1.2 (2021-02-11)

• Alpha-1 release.

0.0.1 (2021-01-20)

• First release on PyPI.
S

snntorch._neurons.alpha, 15
snntorch._neurons.lapicque, 17
snntorch._neurons.leaky, 20
snntorch._neurons.neurons, 38
snntorch._neurons.rleaky, 22
snntorch._neurons.rsynaptic, 25
snntorch._neurons.sconv2dlstm, 29
snntorch._neurons.slstm, 32
snntorch._neurons.synaptic, 35
snntorch.backprop, 40
snntorch.functional.acc, 45
snntorch.functional.loss, 45
snntorch.functional.probe, 54
snntorch.functional.quant, 53
snntorch.functional.reg, 52
snntorch.spikegen, 60
snntorch.spikeplot, 72
snntorch.surrogate, 82
snntorch.utils, 92
Python Module Index
INDEX

A

accuracy_rate() (in module snntorch.functional.acc), 45
accuracy_temporal() (in module snntorch.functional.acc), 45
Alpha (class in snntorch._neurons.alpha), 15
animate() (snntorch.spikeplot.Camera method), 72
animator() (in module snntorch.spikeplot), 72
ATan (class in snntorch.surrogate), 82
atan() (in module snntorch.surrogate), 91
AttributeMonitor (class in snntorch.functional.probe), 54
BaseMonitor (class in snntorch.functional.probe), 55
BPTT() (in module snntorch.backprop), 40
B

backward() (snntorch._neurons.neurons.SpikingNeuron.ATan static method), 39
backward() (snntorch.functional.loss.SpikeTime.FirstSpike static method), 45
backward() (snntorch.functional.loss.SpikeTime.MultiSpike static method), 46
backward() (snntorch.functional.loss.SpikeTime.Tolerance static method), 47
backward() (snntorch.functional.quant.StateQuant static method), 53
backward() (snntorch.surrogate.ATan static method), 83
backward() (snntorch.surrogate.FastSigmoid static method), 84
backward() (snntorch.surrogate.LeakySpikeOperator static method), 85
backward() (snntorch.surrogate.Sigmoid static method), 86
backward() (snntorch.surrogate.SparseFastSigmoid static method), 87
backward() (snntorch.surrogate.SpikeRateEscape method), 88
backward() (snntorch.surrogate.StochasticSpikeOperator static method), 89
backward() (snntorch.surrogate.StraightThroughEstimator static method), 90
backward() (snntorch.surrogate.Triangular static method), 90
BaseMonitor (class in snntorch.functional.probe), 55

C

camera (class in snntorch.spikeplot), 72
ce_count_loss (class in snntorch.functional.loss), 47
ce_max_membrane_loss (class in snntorch.functional.loss), 48
ce_rate_loss (class in snntorch.functional.loss), 49
ce_temporal_loss (class in snntorch.functional.loss), 49
clear_recorded_data()
(ssntorch.functional.probe.BaseMonitor method), 55
create_hook() (snntorch.functional.probe.AttributeMonitor method), 55
create_hook() (snntorch.functional.probe.GradInputMonitor method), 57
create_hook() (snntorch.functional.probe.GradOutputMonitor method), 58
create_hook() (snntorch.functional.probe.InputMonitor method), 59
create_hook() (snntorch.functional.probe.OutputMonitor method), 60

data_subset() (in module snntorch.utils), 92
delta() (in module snntorch.spikegen), 60
detach() (snntorch._neurons.neurons.SpikingNeuron static method), 39
detach_hidden() (snntorch._neurons.alpha.Alpha class method), 39
detach_hidden() (snntorch._neurons.lapicque.Lapicque class method), 19

detach_hidden() (snntorch._neurons.leaky.Leaky class method), 21

detach_hidden() (snntorch._neurons.rleaky.RLeaky class method), 24

detach_hidden() (snntorch._neurons.rsynaptic.RSynaptic class method), 28

detach_hidden() (snntorch._neurons.sconv2dlstm.SConv2dLSTM class method), 31

D

227
<table>
<thead>
<tr>
<th>Method</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>detach_hidden()</td>
<td>snntorch._neurons.slstm.SLSTM class method</td>
<td>34</td>
</tr>
<tr>
<td>detach_hidden()</td>
<td>snntorch._neurons.synaptic.Synaptic class method</td>
<td>37</td>
</tr>
<tr>
<td>disable()</td>
<td>snntorch.functional.probe.BaseMonitor method</td>
<td>55</td>
</tr>
<tr>
<td>DVSGesture</td>
<td>class in snntorch.spikevision.spikedata.dvs_gesture</td>
<td>78</td>
</tr>
<tr>
<td>enable()</td>
<td>snntorch.functional.probe.BaseMonitor method</td>
<td>55</td>
</tr>
<tr>
<td>fast_sigmoid()</td>
<td>in module snntorch.surrogate</td>
<td>91</td>
</tr>
<tr>
<td>FastSigmoid</td>
<td>class in snntorch.surrogate</td>
<td>83</td>
</tr>
<tr>
<td>fire()</td>
<td>snntorch._neurons.neurons.SpikingNeuron method</td>
<td>39</td>
</tr>
<tr>
<td>fire_inhibition()</td>
<td>snntorch._neurons.neurons.SpikingNeuron method</td>
<td>39</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch._neurons.alpha.Alpha method</td>
<td>16</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch._neurons.lapicque.Lapicque method</td>
<td>19</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch._neurons.leaky.Leaky method</td>
<td>21</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch._neurons.neurons.SpikingNeuron.ATan static method</td>
<td>39</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch._neurons.rleaky.RecurrentOneToOne method</td>
<td>25</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch._neurons.rleaky.RLeaky method</td>
<td>24</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch._neurons.rsynaptic.RecurrentOneToOne method</td>
<td>28</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch._neurons.rsynaptic.RSynaptic method</td>
<td>28</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch._neurons.sconv2dlstm.SConv2dLSTM method</td>
<td>32</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch._neurons.slstm.SLSTM method</td>
<td>34</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch._neurons.synaptic.Synaptic method</td>
<td>37</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.functional.loss.SpikeTime method</td>
<td>47</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.functional.loss.SpikeTime.FirstSpike static method</td>
<td>46</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.functional.loss.SpikeTime.MultiSpike static method</td>
<td>46</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.functional.loss.SpikeTime.Tolerance static method</td>
<td>47</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.functional.quant.StateQuant static method</td>
<td>53</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.synaptic.Synaptic method</td>
<td>37</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.surrogate.LeakySpikeOperator static method</td>
<td>85</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.surrogate.Sigmoid static method</td>
<td>86</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.surrogate.SparseFastSigmoid static method</td>
<td>87</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.surrogate.SpikeRateEscape static method</td>
<td>88</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.surrogate.StochasticSpikeOperator static method</td>
<td>89</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.surrogate.StraightThroughEstimator static method</td>
<td>90</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.surrogate.Triangular static method</td>
<td>91</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.surrogate.StateQuant static method</td>
<td>53</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.surrogate.FastSigmoid static method</td>
<td>84</td>
</tr>
<tr>
<td>forward()</td>
<td>snntorch.surrogate.FastSigmoid static method</td>
<td>84</td>
</tr>
<tr>
<td>from_one_hot()</td>
<td>in module snntorch.spikegen</td>
<td>61</td>
</tr>
<tr>
<td>GradInputMonitor</td>
<td>class in snntorch.functional.probe</td>
<td>56</td>
</tr>
<tr>
<td>GradOutputMonitor</td>
<td>class in snntorch.functional.probe</td>
<td>57</td>
</tr>
<tr>
<td>GradInputMonitor</td>
<td>class in snntorch.functional.probe</td>
<td>56</td>
</tr>
<tr>
<td>GradOutputMonitor</td>
<td>class in snntorch.functional.probe</td>
<td>57</td>
</tr>
<tr>
<td>heaviside()</td>
<td>in module snntorch.surrogate</td>
<td>91</td>
</tr>
<tr>
<td>init()</td>
<td>snntorch._neurons.neurons.SpikingNeuron class method</td>
<td>39</td>
</tr>
<tr>
<td>init_alpha()</td>
<td>snntorch._neurons.neurons.LIF static method</td>
<td>38</td>
</tr>
<tr>
<td>init_lapicque()</td>
<td>snntorch._neurons.neurons.LIF static method</td>
<td>38</td>
</tr>
<tr>
<td>init_leaky()</td>
<td>snntorch._neurons.neurons.LIF static method</td>
<td>38</td>
</tr>
<tr>
<td>init_rleaky()</td>
<td>snntorch._neurons.neurons.LIF static method</td>
<td>38</td>
</tr>
<tr>
<td>init_rsynaptic()</td>
<td>snntorch._neurons.neurons.LIF static method</td>
<td>38</td>
</tr>
<tr>
<td>init_sconv2dlstm</td>
<td>snntorch._neurons.sconv2dlstm.SConv2dLSTM static method</td>
<td>32</td>
</tr>
<tr>
<td>init_slstm()</td>
<td>snntorch._neurons.slstm.SLSTM static method</td>
<td>34</td>
</tr>
<tr>
<td>init_synaptic()</td>
<td>snntorch._neurons.neurons.LIF static method</td>
<td>38</td>
</tr>
<tr>
<td>InputMonitor</td>
<td>class in snntorch.functional.probe</td>
<td>58</td>
</tr>
<tr>
<td>instances</td>
<td>snntorch._neurons.neurons.SpikingNeuron attribute</td>
<td>39</td>
</tr>
<tr>
<td>is_enable()</td>
<td>snntorch.functional.probe.BaseMonitor method</td>
<td>56</td>
</tr>
<tr>
<td>l1_rate_sparsity</td>
<td>class in snntorch.functional.reg</td>
<td>52</td>
</tr>
</tbody>
</table>
Index 229

label_to_multi_spike()  
(snntorch.functional.loss.SpikeTime method), 47

label_to_single_spike()  
(snntorch.functional.loss.SpikeTime method), 47

labels_to_spike_times()  
(snntorch.functional.loss.SpikeTime method), 47

Lapicque (class in snntorch._neurons.lapicque), 17

latency() (in module snntorch.spikegen), 61

latency_code() (in module snntorch.spikegen), 62

latency_code_linear() (in module snntorch.spikegen), 63

latency_code_log() (in module snntorch.spikegen), 63

latency_interpolate() (in module snntorch.spikegen), 64

Leaky (class in snntorch._neurons.leaky), 20

LeakySpikeOperator (class in snntorch.surrogate), 84

LIF (class in snntorch._neurons.neurons), 38

LossFunctions (class in snntorch.functional.loss), 45

LSO() (in module snntorch.surrogate), 84

mem_reset() (snntorch._neurons.neurons.SpikingNeuron method), 40

module

snntorch._neurons.alpha, 15
snntorch._neurons.lapicque, 17
snntorch._neurons.leaky, 20
snntorch._neurons.neurons, 38
snntorch._neurons.rleaky, 22
snntorch._neurons.rsynaptic, 25
snntorch._neurons.sconv2dlstm, 29
snntorch._neurons.slstm, 32
snntorch._neurons.synaptic, 35
snntorch.backprop, 40
snntorch.functional.acc, 45
snntorch.functional.loss, 45
snntorch.functional.probe, 54
snntorch.functional.quant, 53
snntorch.functional.reg, 52
snntorch.spikegen, 60
snntorch.spikeplot, 72
snntorch.surrogate, 82
snntorch.utils, 92

mse_count_loss (class in snntorch.functional.loss), 50
mse_membrane_loss (class in snntorch.functional.loss), 50
mse_temporal_loss (class in snntorch.functional.loss), 51

N

NMNIST (class in snntorch.spikevision.spikedata.nmnist), 77

O

OutputMonitor (class in snntorch.functional.probe), 59

R

raster() (in module snntorch.spikeplot), 73
rate() (in module snntorch.spikegen), 65
rate_conv() (in module snntorch.spikegen), 65
rate_interpolate() (in module snntorch.spikegen), 66

RecurrentOneToOne (class in snntorch._neurons.rleaky), 25

remove_hooks() (snntorch.functional.probe.BaseMonitor method), 56
reset() (in module snntorch.utils), 92
reset_dict (snntorch._neurons.neurons.SpikingNeuron attribute), 40
reset_hidden() (snntorch._neurons.alpha.Alpha class method), 16
reset_hidden() (snntorch._neurons.lapicque.Lapicque class method), 19
reset_hidden() (snntorch._neurons.leaky.Leaky class method), 22
reset_hidden() (snntorch._neurons.rleaky.RLeaky class method), 25
reset_hidden() (snntorch._neurons.rsynaptic.RSynaptic class method), 28

reset_hidden() (snntorch._neurons.sconv2dlstm.SConv2dLSTM class method), 32
reset_hidden() (snntorch._neurons.slstm.SLSTM class method), 35
reset_hidden() (snntorch._neurons.synaptic.Synaptic class method), 37
reset_mechanism (snntorch._neurons.neurons.SpikingNeuron property), 40

RLeaky (class in snntorch._neurons.rleaky), 22

RSynaptic (class in snntorch._neurons.rsynaptic), 25

RTRL() (in module snntorch.backprop), 42

S

SConv2dLSTM (class in snntorch._neurons.sconv2dlstm), 29

SFS() (in module snntorch.surrogate), 85

SHD (class in snntorch.spikevision.spikedata.shd), 80

Sigmoid (class in snntorch._neurons.leaky), 20

sigmoid() (in module snntorch.surrogate), 91

SLSTM (class in snntorch._neurons.slstm), 32

snap() (snntorch.spikeplot.Camera method), 72

snntorch._neurons.alpha module, 15

snntorch._neurons.lapicque module, 17
Index