User Manual of SNN simulator SNN-memristor-basedV1.0

Developer: Andrey D. (anddudkin000@gmail.com)

Nov 11, 2023

Table of Contents

1. Installation

User can clone repository to local machine and use all instruments.

```
git clone https://github.com/anddudkin/SNN-memristor-based.git
```

Required packages:

```
Torch 1.8.1
Torchvision 0.9.1
Matplotlib 3.5.2
Tqdm 4.64.1
NumPy 1.21.5
```

2. How to use

1.1 Datasets

Download and initialize dataset. In v1.0 release MNIST is in-build dataset and it can be initialized in 3 different configurations: image size 28x28, 14x14 and 9x9.

To initialize dataset:

1. Call *MNIST_train_test() / MNIST_train_test_14x14() / MNIST_train_test_9x9()* from *datasets.py*

2. Assign generated train and test datasets to the local variables

Example:

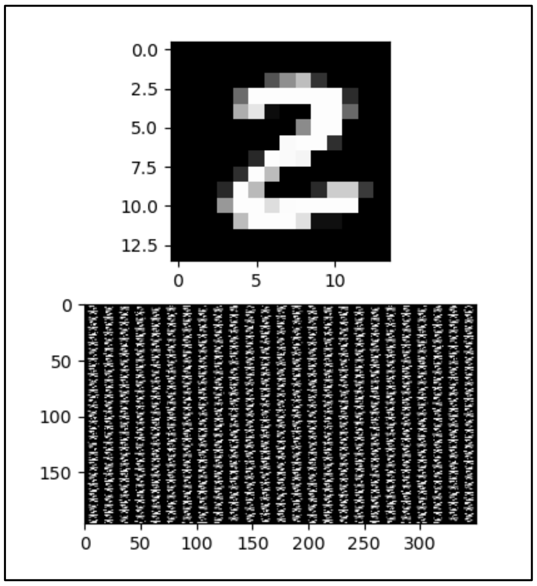
```
data_train = MNIST_train_test_14x14()[0]
data_test = MNIST_train_test_14x14()[1]
```

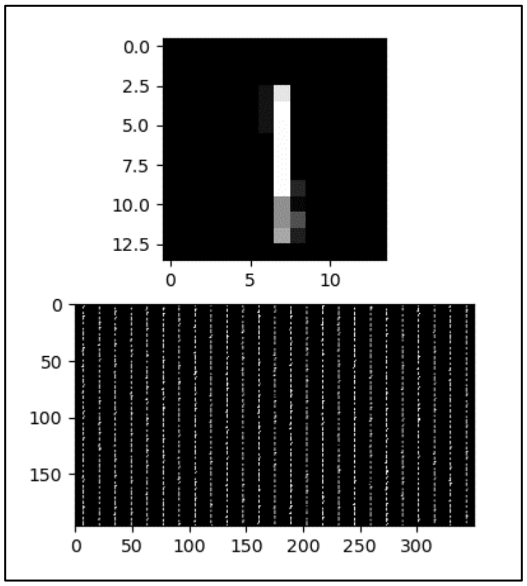
1.1.1 Spike Encoding

To encode images from imported MNIST dataset call *encoding_to_spikes(data, time)* function from datasets.py, and pass image to *data* parameter. Also, it is necessary to specify how much samples (*time* parameter) of the image we want to present during each image presentation.

Example:

```
input_spikes = encoding_to_spikes(data_train[1][0], 350)
```





1.2. Neurons

In v1.0 user can choose between 3 different models of spiking neurons:

- IF neuron
- LIF neuron
- LIF neuron with adaptive threshold

To initialize layer of neurons:

1. Create a new instance of the class *NeuronIF / NeuronLIF / NeuronLifAdaptiveThresh* and assigns this object to the local variable.

Example:

```
out_neurons = NeuronLifAdaptiveThresh(n_neurons_in,
                                      n_neurons_out,
                                      train=True,
                                      U_mem=0,
                                      decay=0.92,
                                      U_tr=20,
                                      U_rest=0,
                                      refr_time=5,
                                      traces=True,
                                      inh=True)
```

1.2.1 Neurons features

IF neuron – Integrate and Fire Neuron

`NeuronIF()` – basic class in NeuronModels.py and Is the parent class for other neuron models

Takes as arguments:

```
n_neurons_in (int) : number of input IF neurons
n_neurons_in (int) : number of output IF neurons
inh (bool)  : activate inhibition or not
traces (bool)  : activate traces or not
train (bool) : train or test
U_tr :  max capacity of neuron (Threshold). If U_mem > U_tr neuron spikes
U_mem :  initialized membrane potentials
U_rest  : membrane potential while resting (refractory), after neuron spikes
refr_time (int) : refractory period time
```

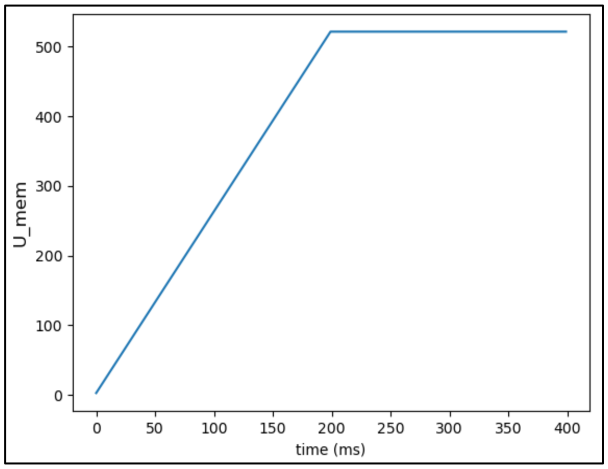
Basic methods of any class of neurons:

`compute_U_mem()` – compute $\Delta Umem$ and update membrane potential.

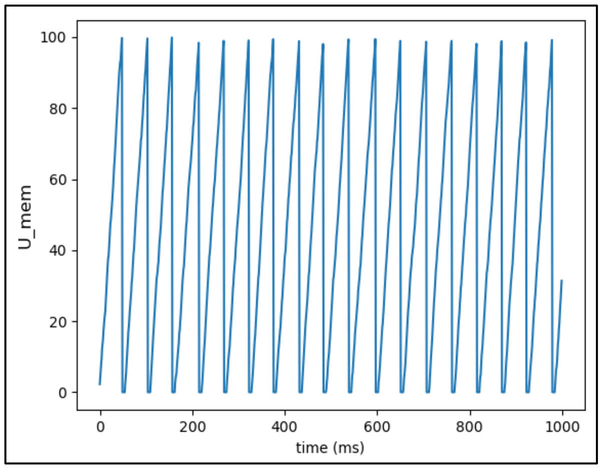
`check_spikes()` – checks if membrane potential crossed membrane threshold and makes neuron spike.

`reset_variables()` – resets variables if needed.

Membrane dynamics shown in the figure below.



DC current to neuron (0 – 200ms)
No current to neuron (200 – 400ms)

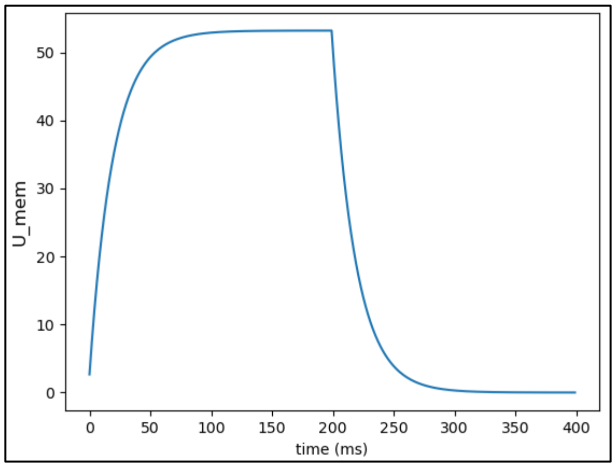


AC current to neuron ≡ spike train

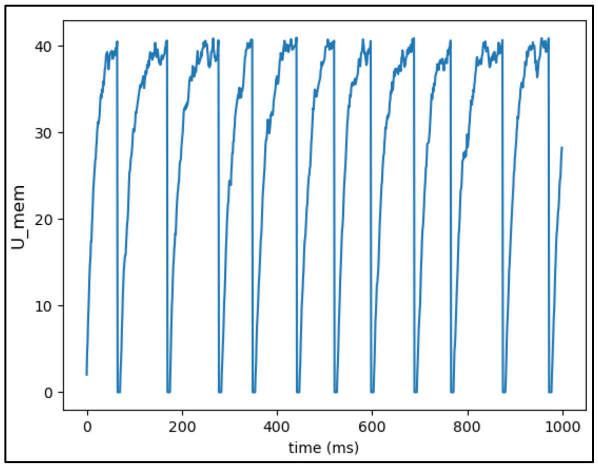
LIF neuron - Leaky Integrate and Fire Neuron

Unlike the IF neuron, the LIF neuron have constant leakage of membrane potential and *NeuronLIF()* class has one more parameter:

`decay (float) : leak of membrane potential`



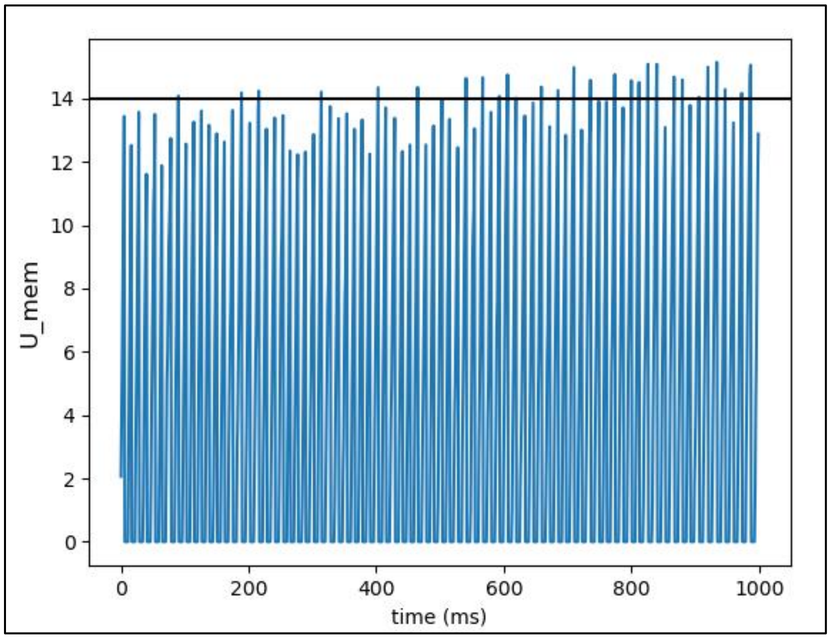
DC current (0 – 200ms)
No current (200 – 400ms)



AC current ≡ spike train

LIF neuron with adaptive threshold - Leaky Integrate and Fire Neuron with adaptive threshold

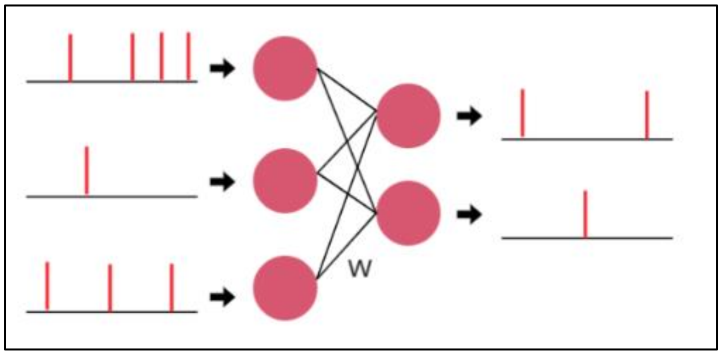
NeuronLifAdaptiveThresh() has the same membrane dynamics as LIF neuron and in addition has adaptive threshold function. This means that the more often the neuron spikes, the value of membrane threshold is getting higher, as shown in figure below. Threshold value exponentially decaying in time.



Black line – default threshold value

1.3 Connections

In v1.0 release only one type of the connection between input and output neurons presented – “all to all” type of connection (each input neuron connected with synapses to all output neurons)



To initialize connections between neurons:

1. Create a new instance of the class *Connections* and assigns this object to the local variable.

2. Call *all_to_all_conn()* method to specify type of connections between neurons.

3. Call *initialize_weights()* method to initialize weights of the synapses.

Example:

```
conn = Connections(n_neurons_in, n_neurons_out, "all_to_all")
conn.all_to_all_conn()
conn.initialize_weights("normal")
```

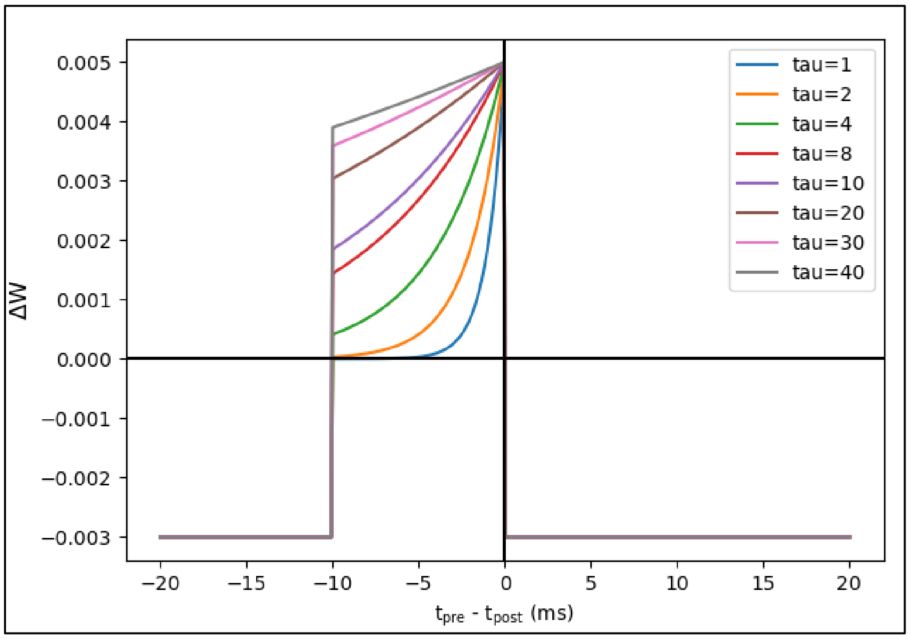
1.4. Learning rules

For faster computation implemented simplified STDP learning rule:

$$\Delta w = A_+ * e^{\frac{t_{pre}-t_{post}}{\tau}} \quad if\ t_{post} > t_{pre}\ and\ \Delta t < t_+$$

$$\Delta w = A_- \quad if\ t_{post} < t_{pre}$$

where $\Delta t = t_{pre} - t_{post}$, Δw denotes the change in the synaptic weight, A_+ and A_- determine the maximum amount of synaptic modification, τ (tau on the pic.) determine the ranges of pre-to-postsynaptic interspike intervals over which synaptic strengthening or weakening occurs.

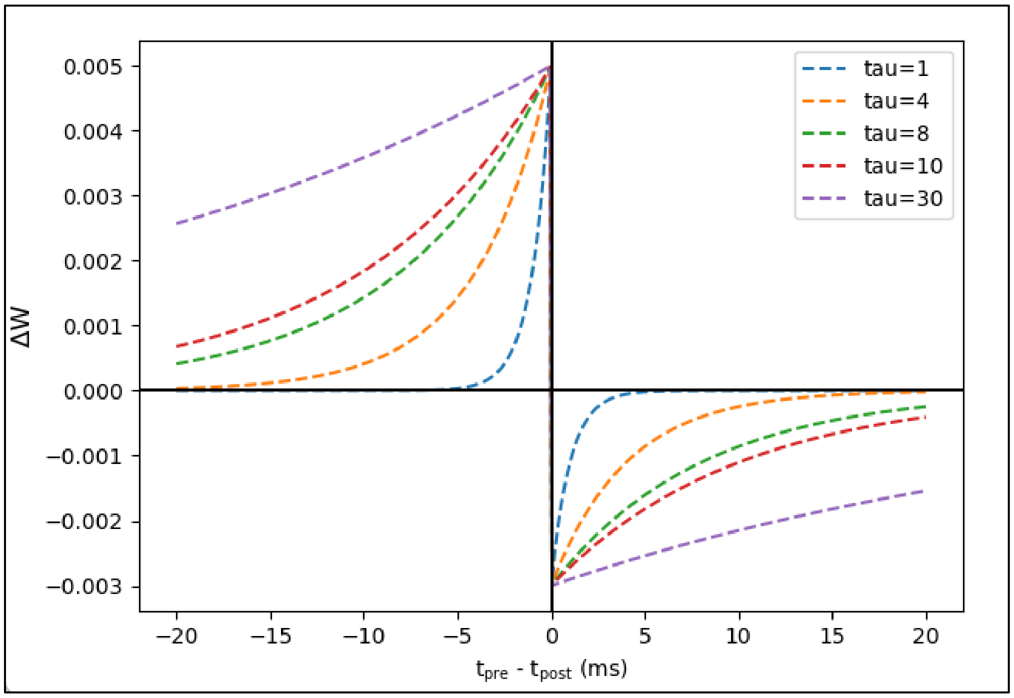


To compute Δw, call *compute_dw()* function from *learning.py* and pass Δt as parameter (Δt can be computed with *Connection* class methods).

For example, a classic STDP rule is given below.

$$\Delta W = A_+ * e^{\frac{t_{pre}-t_{post}}{\tau}} \quad if\ t_{post} > t_{pre}$$

$$\Delta W = -A_- * e^{\frac{-(t_{pre}-t_{post})}{\tau}} \qquad if\ t_{post} < t_{pre}$$



1.5. Labels assignment and accuracy evaluation

This release supports only unsupervised training of spiking neural network with STDP plasticity as local learning rule. Thus, each neuron is assigned a label (digit 0-9) at the end of training based on the neurons overall spiking activity during training.

For tracking spiking activity of each neuron:

1. Create a new instance of the class *MnistAssignment* and assigns this object to the local variable.

```
assig = MnistAssignment(n_neurons_out)
```

2. Call *count_spikes_train(spikes, label)* method after *check_spikes()* method of *NeuronModels* class during training. This method will count how many times each neuron spiked in response for each label during training, so we must pass spikes and label as arguments.

```
assig.count_spikes_train(out_neurons.spikes, data_train[i][1])
```

3. After the end of the training, we call *get_assigment()* method to get label assignments for each neuron. List of assignments will be stored in *assig.assignments* attribute of *MnistAssignment* class.

```
assig.get_assigment()
```

For evaluation of trained network accuracy:

1. Create a new instance of the class *MnistEvaluation* and assigns this object to the local variable.

```
evall = MnistEvaluation(n_neurons_out)
```

2. Call *count_spikes (spikes)* method after *check_spikes()* method of *NeuronModels* class during testing. This method will count how many times each neuron spiked during testing, so we must pass spikes as arguments.

```
evall.count_spikes(out_neurons.spikes)
```

3. After image passed to the network, call *conclude(assignments, label)* method to check if neuron with the highest number of spikes corresponds to label of presented image. Method counts how many images were defined correctly and incorrectly.

```
evall.conclude(assig.assignments, data_train[i][1])
```

4. After testing is done, call *final()* method, that will compute network accuracy and print results in console.

```
evall.final()
```

Implemented evaluation scheme is rather simple end can be further developed to show more precise results. For example, we can take into account the average firing rate of each neuron pool and define presented image more accurately.

1.6. Visualization of network information

Visuals.py contains some functions that can help visualize network data. Main plotting library is Matplotlib 3.5.2

1. Function plot_weights_square(n_in, n_out, weights) help to transform weights matrix to its square representation.

Weights matrix, where $x - \mathit{input\ neuron\ index}, y - \mathit{output\ neuron\ index}$:

$$\begin{bmatrix} w_{11} & \cdots & w_{1y} \\ \vdots & \ddots & \vdots \\ w_{x1} & \cdots & w_{xy} \end{bmatrix}$$

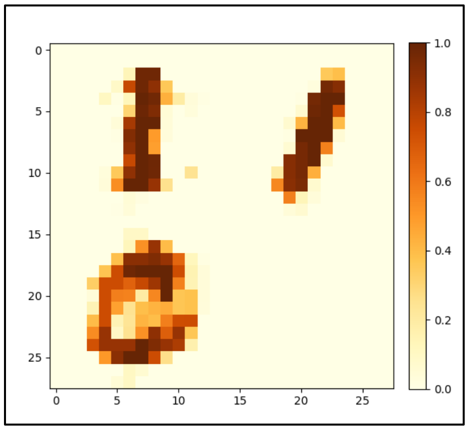
thus, each column represents all weights of corresponding output neuron.

After transformation we can plot weights of each neuron as square images.

Example below shows transformed weights matrix for network with 3 output neurons and 4 input neurons (image size 2x2)

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix} \rightarrow \begin{bmatrix} \begin{bmatrix} w_{11} & w_{21} \\ w_{31} & w_{41} \end{bmatrix} & \begin{bmatrix} w_{12} & w_{22} \\ w_{32} & w_{42} \end{bmatrix} \\ \begin{bmatrix} w_{13} & w_{23} \\ w_{33} & w_{43} \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \end{bmatrix}$$

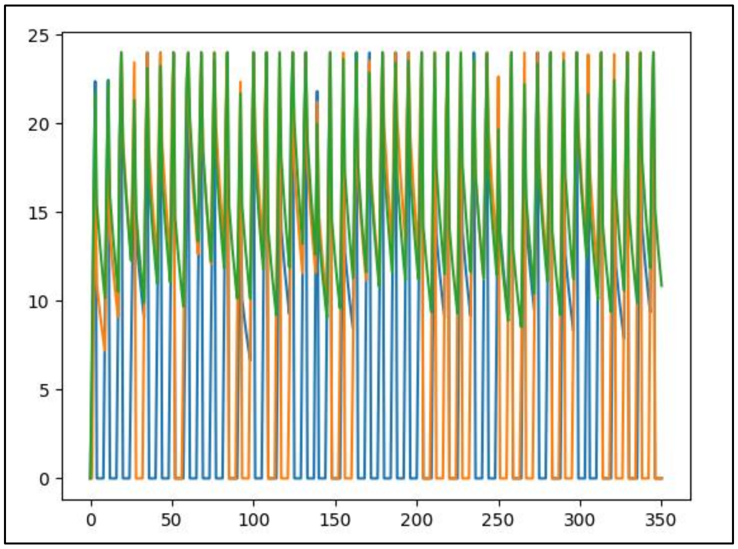
For image with size 14x14 plot will look like this:



2. Function *plot_U_mem(n_neurons_out, U_mem_trace)* can plot dynamic of membrane potential if we pass attribute *U_mem_trace* from *NeuronModels* classes as argument.

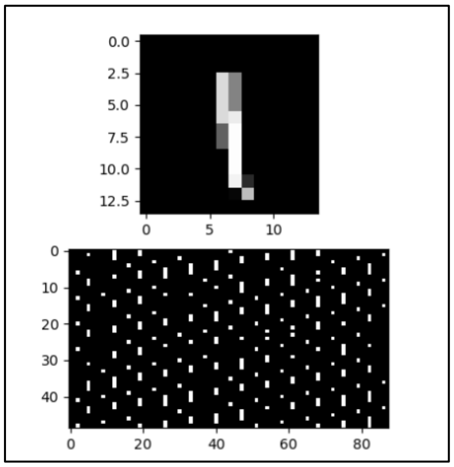
Example:

```
plot_U_mem(n_neurons_out,out_neurons.U_mem_trace)
out_neurons.U_mem_trace= torch.zeros([1, n_neurons_out], dtype=torch.float)
```



3. User can also plot other data with the help of standard Matplotlib modules.

For example, image and generated spike train during training or testing. Code example will be shown in section 3.



1.7. Additional tools

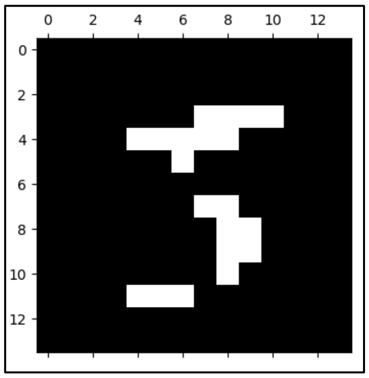
There are some tools in *tools.py,* that user can use to check how different parameters affects neuron behavior.

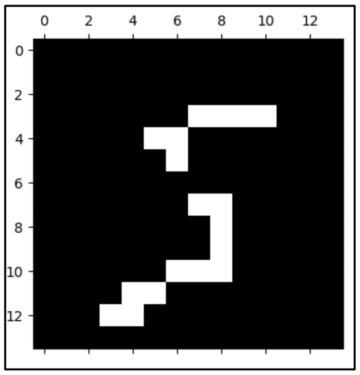
`if_neuron_test()` – simulates one IF neuron with the specified parameters.

`lif_neuron_test()` – simulates one LIF neuron with the specified parameters.

`lif_thresh_neuron_test()` – simulates one LIF neuron with adaptive threshold with the specified parameters.

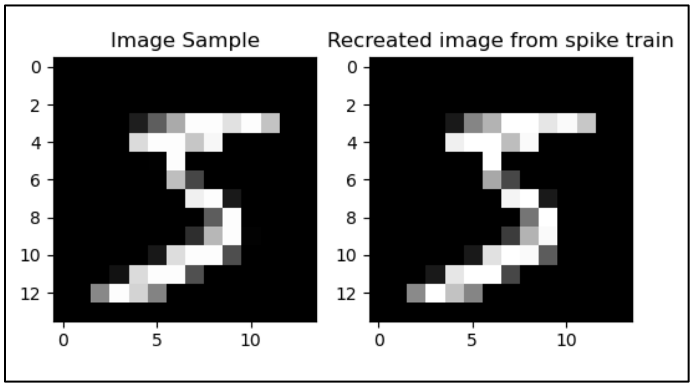
`plot_mnist_test()` – can show animation how selected image sample from MNIST dataset will be transformed to spikes and presented to network on each time step.



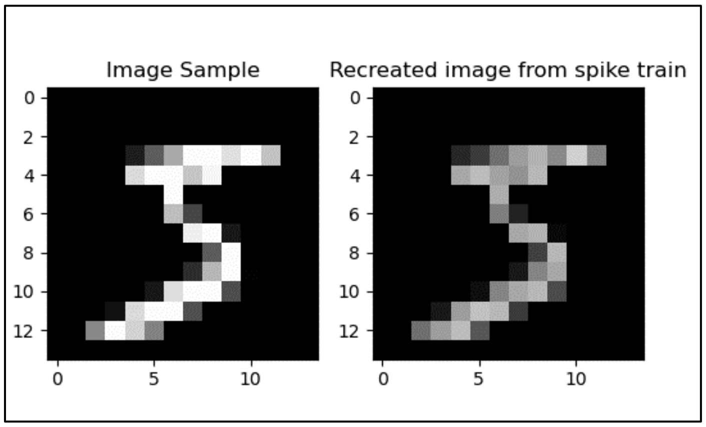


`time_step = 1` `time_step = 2 and so on…`

This function also will plot the original image from dataset and recreated image from generated spike train.



Spike intensity = 100 %



Spike intensity = 70 %

3. Fully connected Spiking Neural Network building example

This example can be found in Examples -> MNIST_example.py

Step by step network building process and final results are shown below.

1. Import required packages

```
import torch
from matplotlib import pyplot as plt
from NeuronModels import NeuronLIF, NeuronIF, NeuronLifAdaptiveThresh
from datasets import MNIST_train_test_14x14, encoding_to_spikes
from topology import Connections
```

2. Setting some basic parameters.

```
n_neurons_out = 50  # number of neurons in input layer
n_neurons_in = 196  # number of output in input layer
n_train = 5000      # number of images for training
n_test = 800        # number of images for testing
time = 350          # time of each image presentation during training
time_test = 200     # time of each image presentation during testing
test = True         # do testing or not
plot = True         # plot graphics or not
```

3. Initializing output neuron layer.

```
out_neurons = NeuronLifAdaptiveThresh(n_neurons_in,
                                      n_neurons_out,
                                      train=True,
                                      U_mem=0,
                                      decay=0.92,
                                      U_tr=20,
                                      U_rest=0,
                                      refr_time=5,
                                      traces=True,
                                      inh=True) # activate literal inhibition
```

4. Initializing connection (synapses).

```
conn = Connections(n_neurons_in, n_neurons_out, "all_to_all")
conn.all_to_all_conn()
conn.initialize_weights("normal")
```

5. Import MNIST datasets and we can limit the number of classes, that we want to use

```
data_train = MNIST_train_test_14x14()[0]
data_test = MNIST_train_test_14x14()[1]
train_labels = [0, 1, 2, 9, 5]
```

6. Initializing MnistAssignments() class

```
assig = MnistAssignment(n_neurons_out)
```

7. Initializing plots (optional).

```
if plot:
    plt.ion()
    fig = plt.figure(figsize=(6, 6))
    ax = fig.add_subplot(111)
    axim = ax.imshow(plot_weights_square(n_neurons_in,
                                         n_neurons_out, conn.weights),
                     cmap='YlOrBr', vmin=0, vmax=1)
    plt.colorbar(axim, fraction=0.046, pad=0.04)

    fig1 = plt.figure(figsize=(5, 5))
    ax2 = fig1.add_subplot(211)
    ax3 = fig1.add_subplot(212)
    axim2 = ax2.imshow(torch.zeros([14, 14]),
                       cmap='gray', vmin=0, vmax=1)
    axim3 = ax3.imshow(torch.zeros([196, 350])[::4, ::4],
                       cmap='gray', vmin=0, vmax=1)
```

8. Construct main training loop

```
for i in tqdm(range(n_train), desc='training', colour='green', position=0):

    if data_train[i][1] in train_labels:

        input_spikes = encoding_to_spikes(data_train[i][0], time)

        if plot:
            axim.set_data(plot_weights_square(n_neurons_in,
                                              n_neurons_out, conn.weights))
            axim2.set_data(torch.squeeze(data_train[i][0]))
            axim3.set_data(input_spikes.reshape(196, 350)[::4, ::4])
            fig.canvas.flush_events()

        for j in range(time):
            out_neurons.compute_U_mem(input_spikes[j].reshape(196), conn.weights)
            out_neurons.check_spikes()
            assig.count_spikes_train(out_neurons.spikes, data_train[i][1])
            conn.update_w(out_neurons.spikes_trace_in,
                          out_neurons.spikes_trace_out, out_neurons.spikes)
```

Inner loop takes each element of generated spike train (*input_spikes*) and presents it to the network. Inside inner loop we call *compute_U_mem()* to update membrane potential of output neurons. Then call *check_spikes()* to check which of neurons spiked and after this calling *count_spikes_train()* to keep track on spikes for further assignment of labels. In the end we call *update_w()* to update weights of connections in accordance with the time of pre- and post-synaptic spikes.

9. In the end of training getting assignments of labels for each neuron

```
assig.get_assigment()
```

10. Initializing MnistEvaluation () class

```
evall = MnistEvaluation(n_neurons_out)
```

11. Disabling the training mode and resetting state variables for more accurate testing

```
out_neurons.train = False
out_neurons.reset_variables(True, True, True)
```

12. Construct testing loop

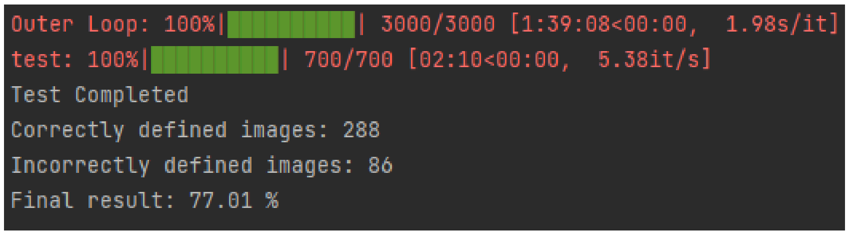
```
if test:
    for i in tqdm(range(n_test), desc='test', colour='green', position=0):

        if data_train[i][1] in train_labels:
            input_spikes = encoding_to_spikes(data_train[i][0], time_test)

            for j in range(time_test):
                out_neurons.compute_U_mem(input_spikes[j].reshape(196),
                                          conn.weights)
                out_neurons.check_spikes()
                evall.count_spikes(out_neurons.spikes)
            evall.conclude(assig.assignments, data_train[i][1])
```

Testing loop is basically the same as training loop, but we don't update weights of connections. We also calling *count_spikes()* and *conclude()* to check if presented image was defined correctly or not.

13. In the final step calling *final(),* to compute overall accuracy of the network.

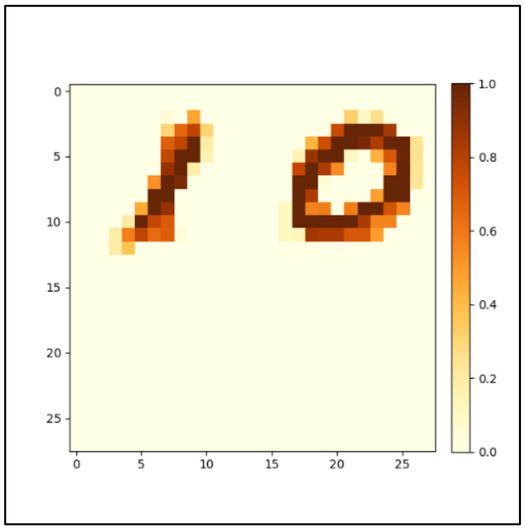
```
evall.final()
```

As the result we can see something like this:

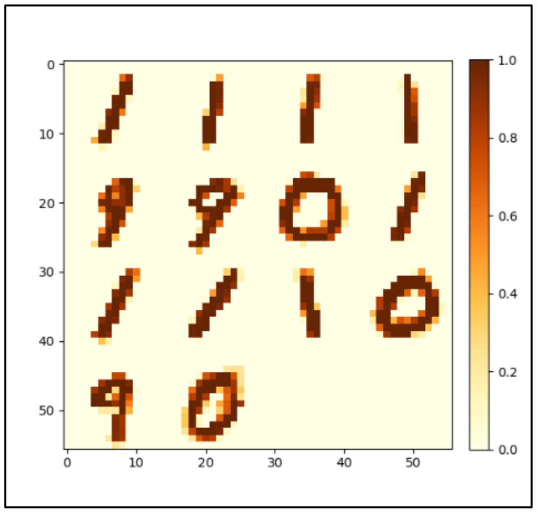


```
Outer Loop: 100%|██████████| 3000/3000 [1:39:08<00:00,  1.98s/it]
test: 100%|██████████| 700/700 [02:10<00:00,  5.38it/s]
Test Completed
Correctly defined images: 288
Incorrectly defined images: 86
Final result: 77.01 %
```

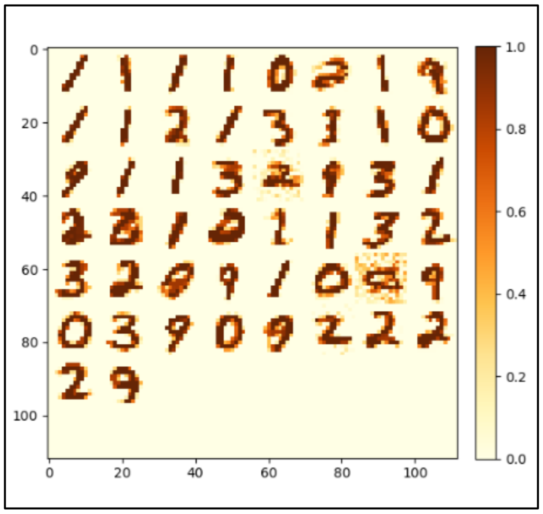
3.1 Conclusion



In classification of only two digits (0,1) this network model with 2 output neurons has shown 100 % accuracy and no problems have been detected.



In classification of three digits (0,1,9) this network model with 14 output neurons has shown 96 % accuracy.



In classification of five digits this network model with 50 output neurons has shown 82 % accuracy.

Constructed network in MNIST_example.py is rather balanced and doesn’t show overexcitement. However, further improvement is required in terms of parameter optimization if we want to see better accuracy, what can be achieved, for example, with the help of a genetic algorithm and bagging technique.