Self-Organizing Recurrent Neural Network (SORN)

Saranraj Nambusubramaniyan

May 16, 2022

GETTING STARTED

1 Installation **3**
- 1.1 Dependencies . . . 3

2 Usage **5**
- 2.1 Plasticity Phase . . . 5
- 2.2 Training phase . . . 7
- 2.3 Freeze plasticity . . . 7
- 2.4 Network Output Descriptions . . . 8
- 2.5 Colaboratory Notebook . . . 8
- 2.6 Usage with OpenAI gym . . . 8
- 2.7 Plotting functions . . . 10
- 2.8 Statistics and Analysis functions . . . 17

3 Reference **19**
- 3.1 SORN Network . . . 19
- 3.2 Utility Functions . . . 26

4 Contributions **33**

5 Cite Package **35**

6 Cite Paper **37**

7 MIT License **39**

8 Contact **41**

9 Indices and tables **43**

Self-Organizing Recurrent Neural (SORN) networks are a class of reservoir computing models build based on plasticity mechanisms in biological brain. Recent studies on SORN shows that such models can mimic neocortical circuit's ability of learning and adaptation through neuroplasticity mechanisms. Structurally, unlike other liquid state models, SORN consists of pool of excitatory neurons and small population of inhibitory neurons. First such network was introduced with three fundamental plasticity mechanisms found in neocortex, namely Spike timing dependent plasticity (STDP), intrinsic plasticity (IP) and Synaptic scaling (SS). Spike Timing-Dependent Plasticity or Hebbian Learning with positive feedback (rapid cycle of synaptic potentials) selectively strengthens correlated synapses and weaken the uncorrelated. Such activity dependent rules lead to Long Time Potentiation (LTP) and Long Time Depression (LTD).

Biologically, both LTP and LDP are assumed to possess substrates of learning and memory at the cellular level of neocortex. However, in dynamical systems, such phenomena will drive the network either towards the state of bursting activity in case of LTP or towards state of attenuation due to LTD. These destabilizing influences of STDP are counteracted by homeostatic plasticity mechanisms. Homeostatic mechanisms are a set of negative feedback (action potential suppressing) regulatory mechanisms that scales incoming synaptic strengths and balances neuronal activity through synaptic normalization and intrinsic plasticity. Experimental evidences also prove that synaptic scaling found to balance the activity between excitatory and inhibitory neurons in-vivo. Together, they maintain the overall activity of network within subcritical range, despite the network being driven by positive feedback from fast Hebbian plasticity.

In recent proposed models, SORN is extended with two more plasticity mechanisms, inhibitory spike timing dependent plasticity and structural plasticity. While connections between excitatory neurons (E-E) subjected to STDP rules, connections from inhibitory population to excitatory populations(E-I) are regulated by iSTDP. Structural plasticity, generates new connections constantly at a smaller rate between unconnected synapses. Many studies argued that, such structural changes induce neuronal morphogenesis which leads to network re-organization with functional consequences over learning and memory. The mathematical descriptions of plasticity mechanisms proposed in SORN simplifies the structural and functional connectivity mechanisms that resembles information processing, learning and memory phenomena that occur in neuro-synapses of neocortex region. Recent experimental evidences confirm that SORN outperforms other static reservoir networks in spatio-temporal tasks and maintains the dynamics of the network in subcritical state suitable for learning. Further research on such network mechanisms unravels the underlying features of synaptic connections and network activity in real cortical circuits. Hence investigating the characteristics of SORN and extending its structural and functional attributes by replicating the recent findings in neural connectomics may reveal the dominating principles of self-organization and self-adaptation in neocortical circuits at microscopic level. Moreover, characterizing these mechanisms individually at that level may also help us to understand some fundamental aspects of brain networks at mesoscopic and macroscopic scales.

CHAPTER

ONE

INSTALLATION

Install using *pip*

pip install sorn

or

To install the latest version from the development branch

pip install git+https://github.com/Saran-nns/sorn

1.1 Dependencies

SORN supports Python 3.5+ ONLY. For older Python versions please use the official Python client

To install all optional dependencies run

pip install 'sorn[all]'

CHAPTER

TWO

USAGE

2.1 Plasticity Phase

```
import sorn
from sorn import Simulator
import numpy as np

# Sample input
num_features = 10
time_steps = 200
inputs = np.random.rand(num_features,time_steps)

# Simulate the network with default hyperparameters under gaussian white noise
state_dict, E, I, R, C = Simulator.simulate_sorn(inputs = inputs, phase='plasticity',
                                                matrices=None, noise = True,
                                                time_steps=time_steps)
```

```
Network Initialized
Number of connections in Wee 3909 , Wei 1574, Wie 8000
Shapes Wee (200, 200) Wei (40, 200) Wie (200, 40)
```

The default values of the network hyperparameters are,

Table 1: Hyperparameters of the network and default values

Keyword argument	Value	Description
ne	200	Number of Encitatory neurons in the reservoir
nu	10	Number of Input neurons in the reservoir
network_type_ee	"Sparse"	*Sparse* or *Dense* connectivity between Excitatory neurons
network_type_ie	"Dense"	*Sparse* or *Dense* connectivity from Excitatory to Inhibitory neurons
network_type_ei	"Sparse"	*Sparse* or *Dense* connectivity from Inhibitory to Excitatory neurons
lambda_ee	20	% of connections between neurons in Excitatory pool
lambda_ei	40	% of connections from Inhibitory to Excitatory neurons
lambda_ie	100	% of connections from Excitatory to Inhibitory neurons
eta_stdp	0.004	Hebbian Learning rate for connections between excitatory neurons
eta_inhib	0.001	Hebbian Learning rate for connections from Inhibitory to Excitatory neurons
eta_ip	0.01	Intrinsic plasticity learning rate
te_max	1.0	Maximum excitatory neuron threshold value
ti_max	0.5	Maximum inhibitory neuron threshold value
ti_min	0.0	Minimum inhibitory neuron threshold value
te_min	0.0	Minimum excitatory neuron threshold value
mu_ip	0.01	Target mean firing rate of excitatory neuron
sigma_ip	0.0	Standard deviation of firing rate of excitatory neuron

2.1.1 Override the default hyperparameters and simulate new SORN model

```
# Sample input
num_features = 5
time_steps = 1000
inputs = np.random.rand(num_features,time_steps)

state_dict, E, I, R, C = Simulator.simulate_sorn(inputs = inputs, phase='plasticity',
                                                matrices=None, noise = True,
                                                time_steps=time_steps,
                                                ne = 100, nu=num_features,
                                                lambda_ee = 10, eta_stdp=0.001)
```

```
Network Initialized
Number of connections in Wee 959 , Wei 797, Wie 2000
Shapes Wee (100, 100) Wei (20, 100) Wie (100, 20)
```

2.2 Training phase

```
from sorn import Trainer
# NOTE: During training phase, input to `sorn` should have second (time) dimension set to␣
↪1. ie., input shape should be (input_features,1).

inputs = np.random.rand(num_features,1)

# SORN network is frozen during training phase
state_dict, E, I, R, C = Trainer.train_sorn(inputs = inputs, phase='training',
                                            matrices=state_dict, noise= False,
                                            time_steps=1,
                                            ne = 100, nu=num_features,
                                            lambda_ee = 10, eta_stdp=0.001 )
```

2.3 Freeze plasticity

To turn off any plasticity mechanisms during simulation or training phase, use freeze argument. For example to stop intrinsic plasticity during simulation phase,

```
# Sample input
num_features = 10
time_steps = 20
inputs = np.random.rand(num_features,time_steps)

state_dict, E, I, R, C = Simulator.simulate_sorn(inputs = inputs, phase='plasticity',
                                                matrices=None, noise = True,
                                                time_steps=time_steps, ne = 50,
                                                nu=num_features, freeze=['ip'])
```

To train the above model under plasticity mechanisms except ip and istdp, use freeze argument

```
state_dict, E, I, R, C = Trainer.train_sorn(inputs = inputs, phase='plasticity',
                                            matrices=state_dict, noise= False,
                                            time_steps=1,
                                            ne = 50, nu=num_features,
                                            freeze=['ip','istdp'])
```

To train the above model with all plasticity mechanisms frozen , change the phase argument value to training

```
state_dict, E, I, R, C = Trainer.train_sorn(inputs = inputs, phase='training',
                                            matrices=state_dict, noise= False,
                                            time_steps=1,
                                            ne = 50, nu=num_features)
```

The other options are,

stdp - Spike Timing Dependent Plasticity

ss - Synaptic Scaling

sp - Structural Plasticity

istdp - Inhibitory Spike Timing Dependent Plasticity

Note: If you pass all above options to freeze, then the network will behave as Liquid State Machine(LSM) i,e., the connection strengths and thresholds remains fixed at the random intial state.

2.4 Network Output Descriptions

state_dict - Dictionary of connection weights (*Wee*,`Wei`,`Wie`) ,

Excitatory network activity (X),

Inhibitory network activities(Y),

Threshold values (*Te*,`Ti`)

E - Excitatory network activity of entire simulation period

I - Inhibitory network activity of entire simulation period

R - Recurrent network activity of entire simulation period

C - Number of active connections in the Excitatory pool at each time step

2.5 Colaboratory Notebook

Sample simulation and training runs with few plotting functions are found at Colab

2.6 Usage with OpenAI gym

2.6.1 Cartpole balance problem

```
from sorn import Simulator, Trainer
import gym

# Hyperparameters
NUM_EPISODES = int(2e6)
NUM_PLASTICITY_EPISODES = 20

LEARNING_RATE = 0.0001 # Gradient ascent learning rate
GAMMA = 0.99 # Discounting factor for the Rewards

# Open AI gym; Cartpole Environment
env = gym.make('CartPole-v0')
action_space = env.action_space.n

# SORN network parameters
ne = 50
nu = 4
# Init SORN using Simulator under random input;
state_dict, E, I, R, C = Simulator.simulate_sorn(inputs = np.random.randn(4,1),
                                                 phase ='plasticity',
                                                 time_steps = 1,
                                                 noise=False,
```

(continues on next page)

(continued from previous page)

```
                                                     ne = ne, nu=nu)

w = np.random.rand(ne, 2) # Output layer weights

# Policy
def policy(state,w):
    "Implementation of softmax policy"
    z = state.dot(w)
    exp = np.exp(z)
    return exp/np.sum(exp)

# Vectorized softmax Jacobian
def softmax_grad(softmax):
    s = softmax.reshape(-1,1)
    return np.diagflat(s) - np.dot(s, s.T)

for EPISODE in range(NUM_EPISODES):

    # Environment observation;
    # NOTE: Input to sorn should have time dimension. ie., input shape should be (input_
↪features,time_steps)
    state = env.reset()[:, None] # (4,) --> (4,1)

    grads = [] # Episode log policy gradients
    rewards = [] # Episode rewards

    # Keep track of total score
    score = 0

    # Play the episode
    while True:

      # env.render() # Uncomment to see your model train in real time (slow down␣
↪training progress)
      if EPISODE < NUM_PLASTICITY_EPISODES:

        # Plasticity phase
        state_dict, E, I, R, C = Simulator.simulate_sorn(inputs = state, phase =
↪'plasticity',
                                                     matrices = state_dict, time_
↪steps = 1,
                                                     ne = ne, nu=nu,
                                                     noise=False)

      else:
        # Training phase with frozen reservoir connectivity
        state_dict, E, I, R, C = Trainer.train_sorn(inputs = state, phase = 'training',
                                                 matrices = state_dict, time_steps = 1,
                                                 ne = ne, nu=nu,
                                                 noise= False)

      # Feed E as input states to your RL algorithm, below goes for simple policy␣
↪gradient algorithm
```

(continues on next page)

(continued from previous page)

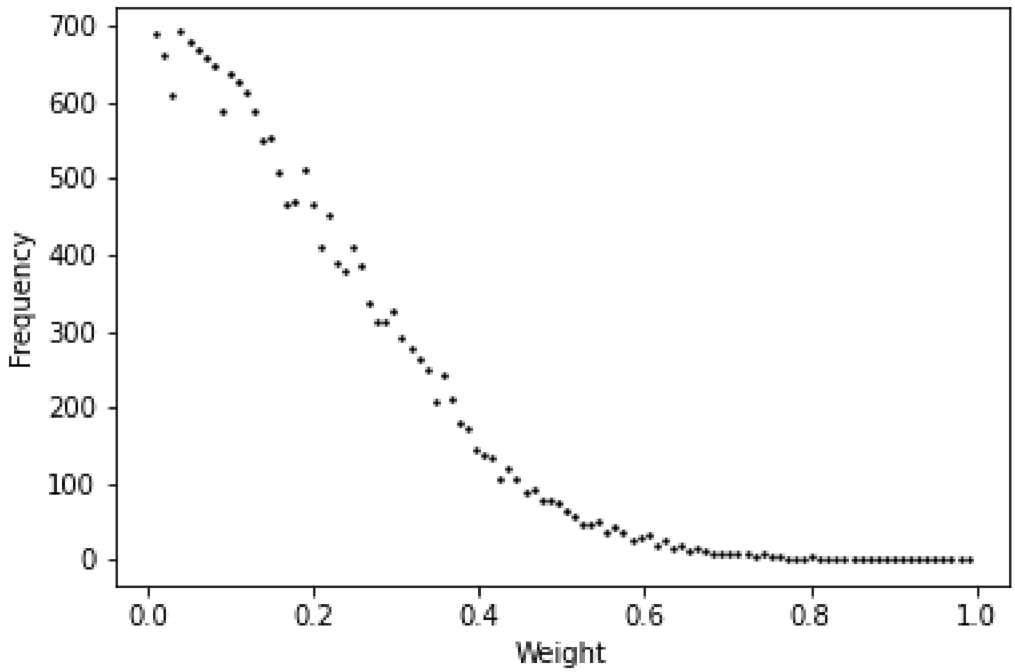
```
        # Sample policy w.r.t excitatory states and take action in the environment
        probs = policy(np.asarray(E),w)
        action = np.random.choice(action_space,p=probs[0])
        state,reward,done,_ = env.step(action)
        state = state[:,None]

        # COMPUTE GRADIENTS BASED ON YOUR OBJECTIVE FUNCTION;
        # Sample computation of policy gradient objective function
        dsoftmax = softmax_grad(probs)[action,:]
        dlog = dsoftmax / probs[0,action]
        grad = np.asarray(E).T.dot(dlog[None,:])
        grads.append(grad)
        rewards.append(reward)
        score+=reward

        if done:
            break

    # OPTIMIZE OUTPUT LAYER WEIGHTS `w` BASED ON YOUR OPTIMIZATION METHOD;
    # Below is a sample of weight update based on gradient ascent(maximize cumulative␣
↪reward) method for temporal difference learning
    for i in range(len(grads)):

        # Loop through everything that happened in the episode and update towards the␣
↪log policy gradient times future reward
        w += LEARNING_RATE * grads[i] * sum([ r * (GAMMA ** r) for t,r in␣
↪enumerate(rewards[i:])])
    print('Episode %s  Score %s' %(str(EPISODE),str(score)))
```

There are several neural data analysis and visualization methods inbuilt with *sorn* package. Sample call for few plotting and statistical methods are shown below;

2.7 Plotting functions

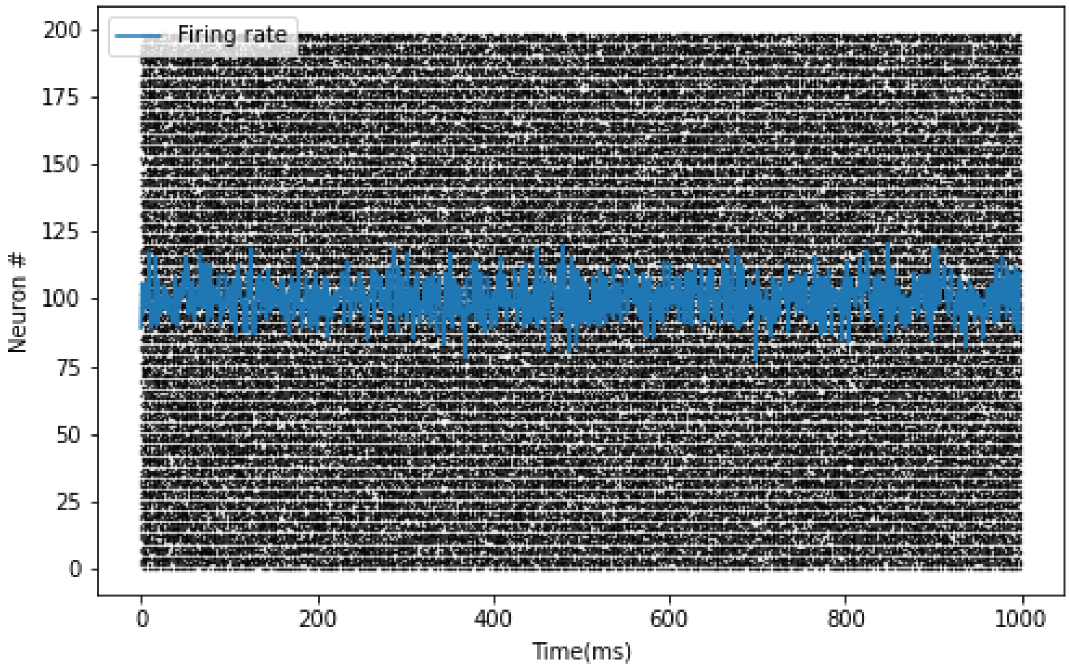
2.7.1 Plot weight distribution in the network

```
from sorn import Plotter
# For example, the network has 200 neurons in the excitatory pool.
Wee = np.random.randn(200,200) # Note that generally Wee is sparse
Wee=Wee/Wee.max() # state_dict['Wee'] returned by the SORN is already normalized
Plotter.weight_distribution(weights= Wee, bin_size = 5, savefig = True)
```



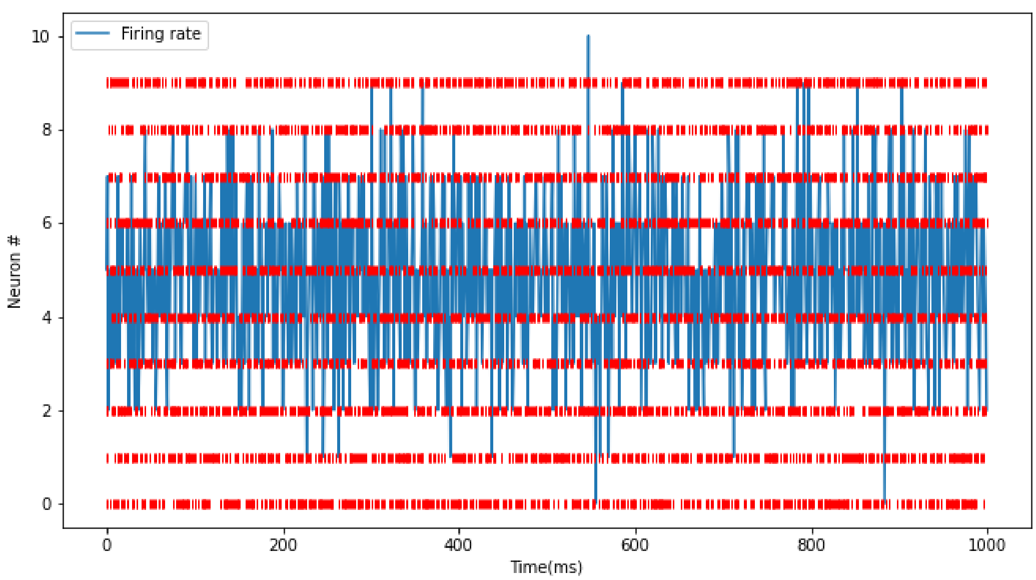
2.7.2 Plot Spike train

```
E = np.random.randint(2, size=(200,1000)) # For example, activity of 200 excitatory␣
↪neurons in 1000 time steps
Plotter.scatter_plot(spike_train = E, savefig=True)
```



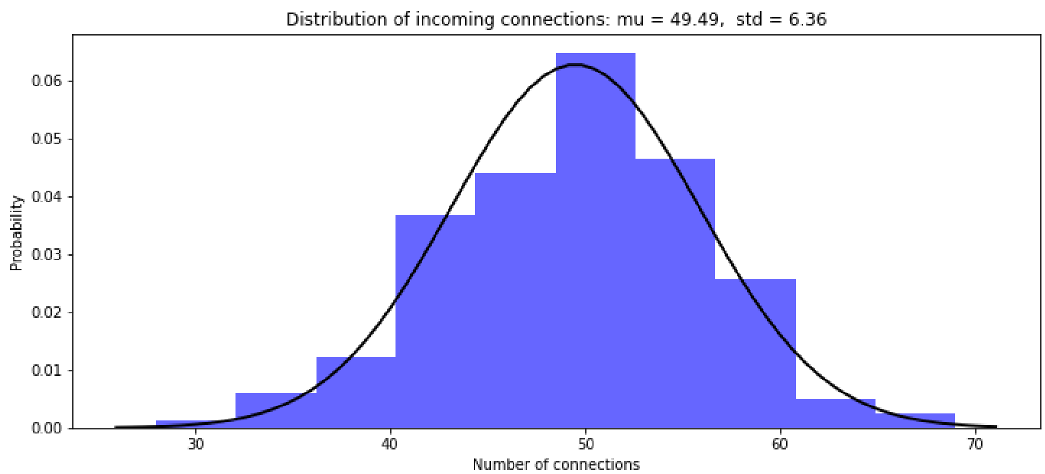
2.7.3 Raster plot of Spike train

```
# Raster plot of activity of only first 10 neurons in the excitatory pool
Plotter.raster_plot(spike_train = E[:,0:10], savefig=True)
```



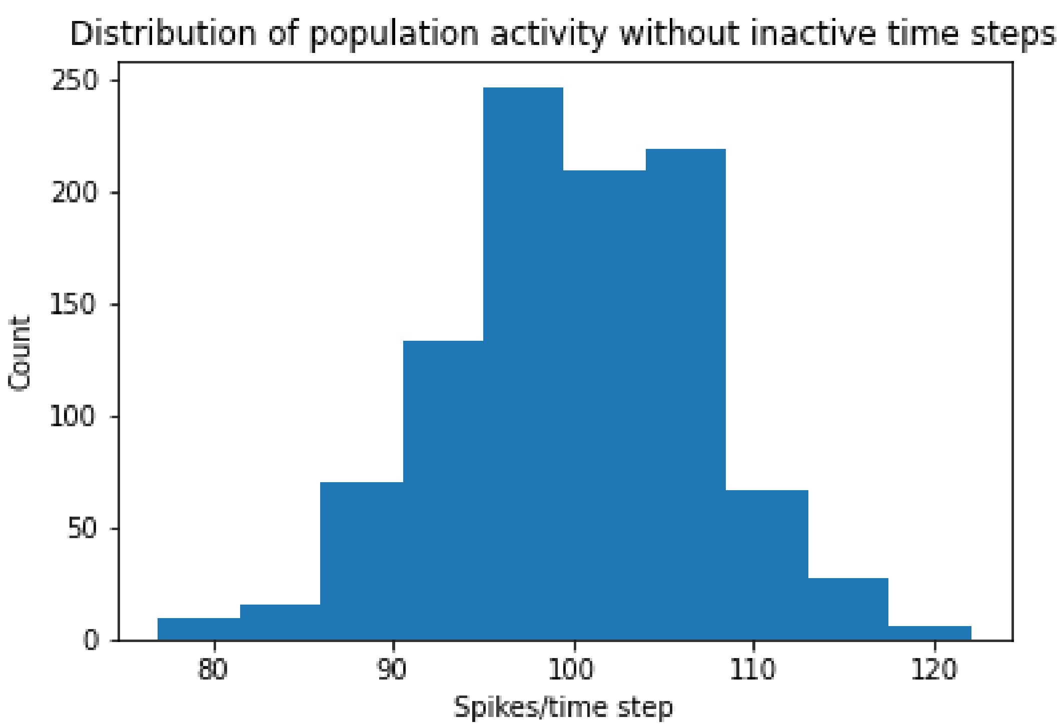
2.7.4 Distribution of presynaptic connections

```
# Histogram of number of presynaptic connections per neuron in the excitatory pool
Plotter.hist_incoming_conn(weights=Wee, bin_size=10, histtype='bar', savefig=True)
```



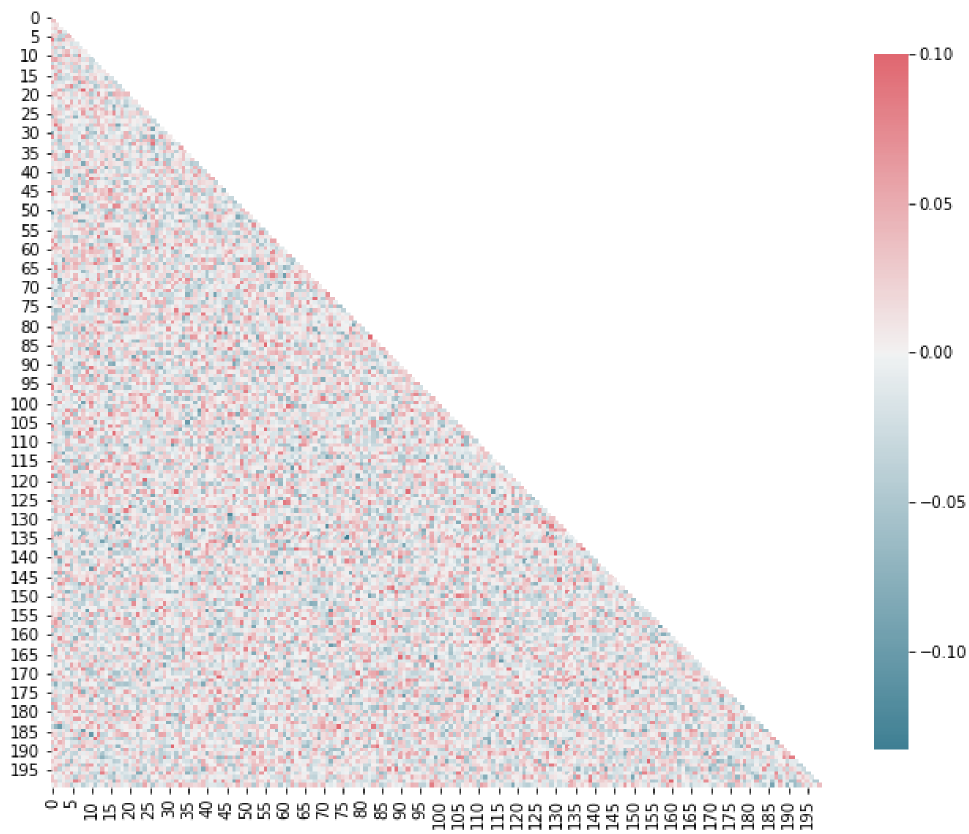
2.7.5 Distribution of firing rate of the network

```
Plotter.hist_firing_rate_network(E, bin_size=10, savefig=True)
```



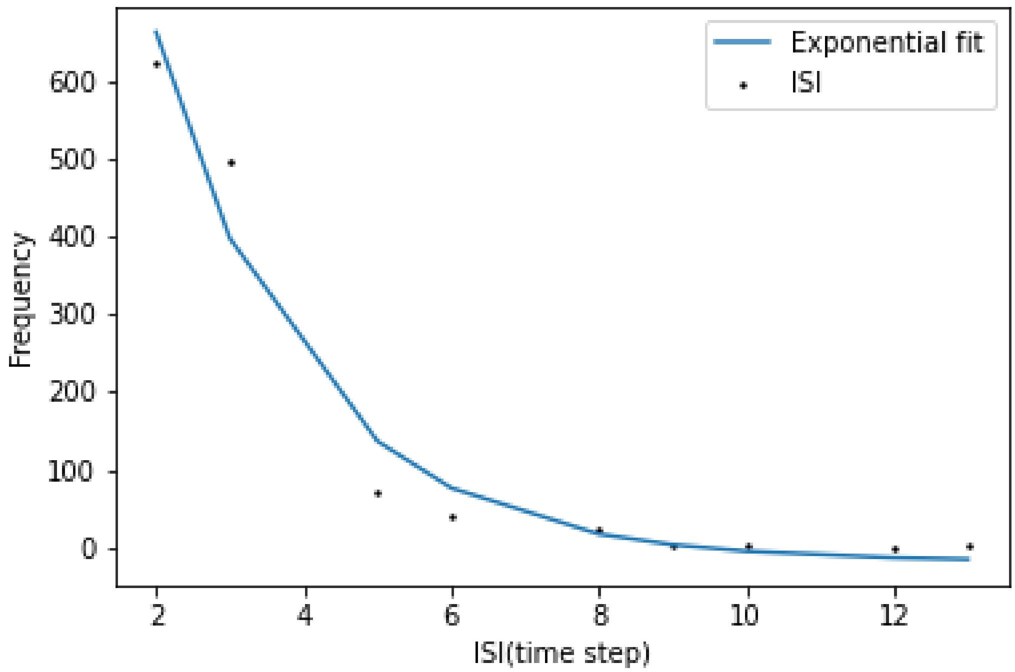
2.7.6 Plot pearson correlation between neurons

```
from sorn import Statistics
avg_corr_coeff,_ = Statistics.avg_corr_coeff(E)
Plotter.correlation(avg_corr_coeff,savefig=True)
```



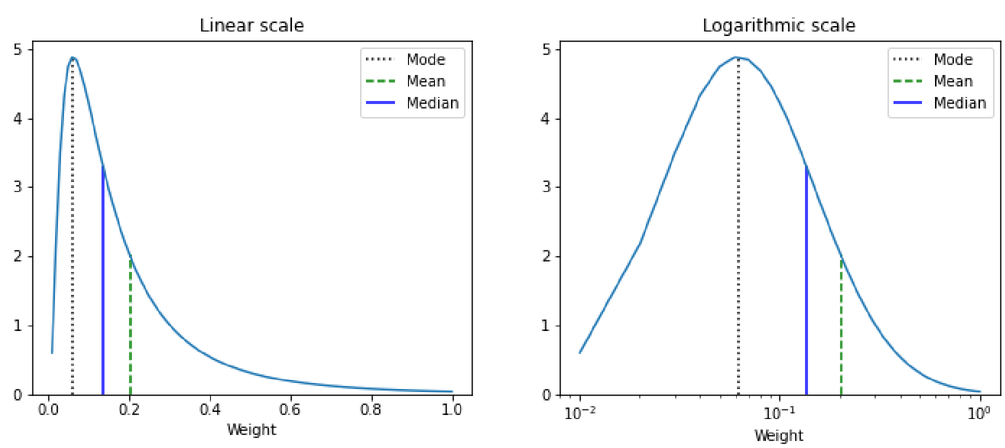
2.7.7 Inter spike intervals

```
# Inter spike intervals with exponential curve fit for neuron 1 in the Excitatory pool
Plotter.isi_exponential_fit(E,neuron=1,bin_size=10, savefig=True)
```



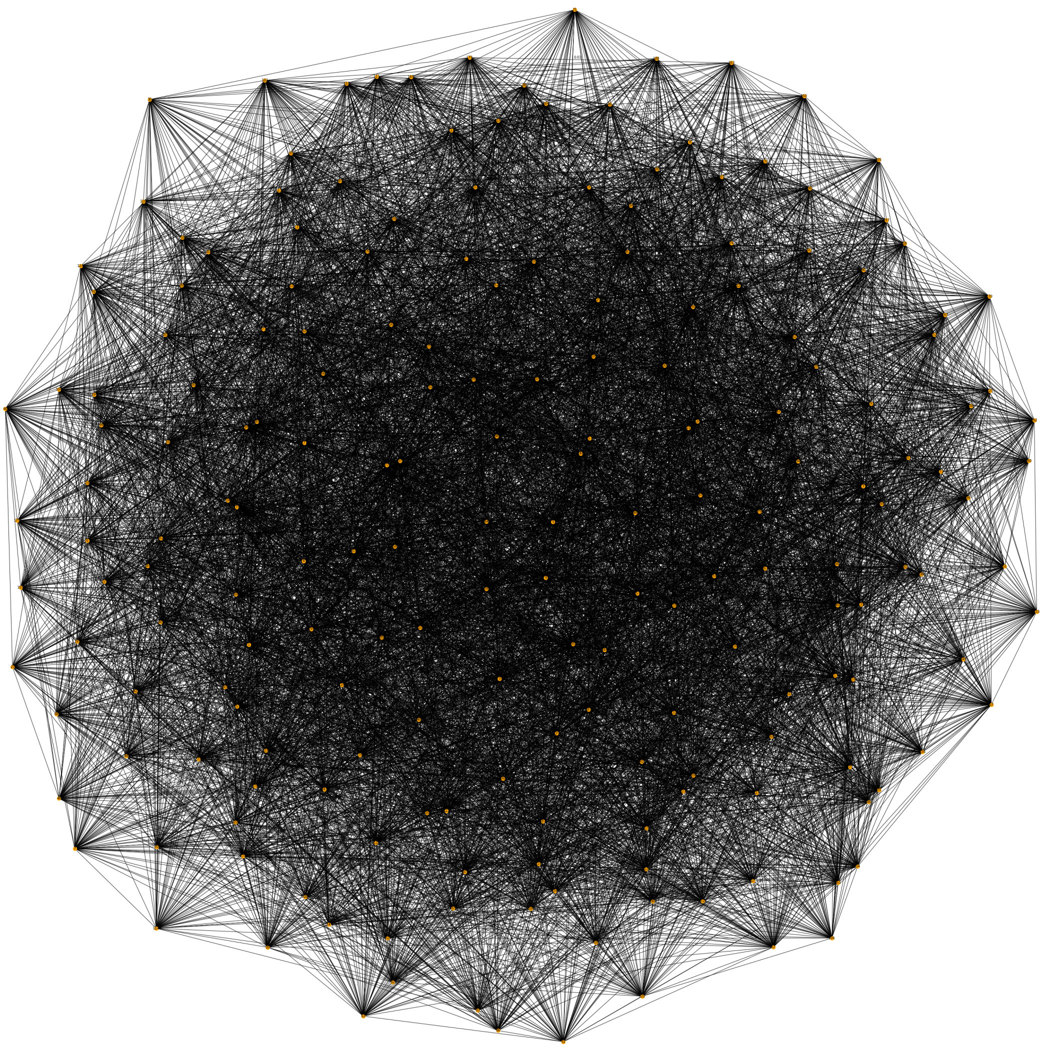
2.7.8 Linear and Lognormal curve fit of Synaptic weights

```
# Distribution of connection weights in linear and lognormal scale
Plotter.linear_lognormal_fit(weights=Wee,num_points=100, savefig=True)
```



2.7.9 Network plot

```
# Draw network connectivity using the pearson correlation function between neurons in
↪the excitatory pool
Plotter.plot_network(avg_corr_coeff,corr_thres=0.01,fig_name='network.png')
```



2.8 Statistics and Analysis functions

2.8.1 t-lagged auto correlation between neural activity

```
from sorn import Statistics
pearson_corr_matrix = Statistics.autocorr(firing_rates = [1,1,5,6,3,7], t= 2)
```

2.8.2 Fano factor

```
# To verify poissonian process in spike generation of neuron 10
mean_firing_rate, variance_firing_rate, fano_factor = Statistics.fanofactor(spike_train=␣
↪E,
                                                                            neuron = 10,
                                                                            window_size = 10)
```

2.8.3 Spike Source Entropy

```
# Measure the uncertainty about the origin of spike from the network using entropy
sse = Statistics.spike_source_entropy(spike_train= E, num_neurons=200)
```

CHAPTER

THREE

REFERENCE

3.1 SORN Network

3.1.1 Sorn

The following methods are available via `SORN`

class `sorn.sorn.`**`Sorn`**

This class wraps initialization of the network and its parameters

`eta_inhib = 0.001`

`eta_ip = 0.01`

`eta_stdp = 0.004`

static **`initialize_activity_vector`**(*ne: int*, *ni: int*)

Initialize the activity vectors X and Y for excitatory and inhibitory neurons

Parameters

- **ne** (*int*) – Number of excitatory neurons
- **ni** (*int*) – Number of inhibitory neurons

Returns Array of activity vectors of excitatory population y (array): Array of activity vectors of inhibitory population

Return type x (array)

static **`initialize_threshold_matrix`**(*te_min: float*, *te_max: float*, *ti_min: float*, *ti_max: float*)

Initialize the threshold for excitatory and inhibitory neurons

Parameters

- **te_min** (*float*) – Min threshold value for excitatory units
- **te_max** (*float*) – Min threshold value for inhibitory units
- **ti_min** (*float*) – Max threshold value for excitatory units
- **ti_max** (*float*) – Max threshold value for inhibitory units

Returns Threshold values for excitatory units ti (array): Threshold values for inhibitory units

Return type te (array)

static initialize_weight_matrix(*network_type: str*, *synaptic_connection: str*, *self_connection: str*, *lambd_w: int*)

Wrapper for initializing the weight matrices for SORN

Parameters

- **network_type** (*str*) – Spare or Dense
- **synaptic_connection** (*str*) – EE,EI,IE. Note that Spare connection is defined only for EE connections
- **self_connection** (*str*) – True or False: Synaptic delay or time delay
- **lambd_w** (*int*) – Average number of incoming and outgoing connections per neuron

Returns Array of connection strengths

Return type weight_matrix (array)

lambda_ee = 20

lambda_ei = 40

lambda_ie = 100

mu_ip = 0.1

ne = 200

network_type_ee = 'Sparse'

network_type_ei = 'Sparse'

network_type_ie = 'Dense'

ni = 40

nu = 10

sigma_ip = 0.0

te_max = 1.0

te_min = 0.0

ti_max = 0.5

ti_min = 0.0

3.1.2 Plasticity

The following methods are available via `Plasticity`

class sorn.sorn.**Plasticity**

Instance of class Sorn. Inherits the variables and functions defined in class Sorn. It encapsulates all plasticity mechanisms mentioned in the article. Inherits all attributed from parent class Sorn

static initialize_plasticity()

Initialize weight matrices for plasticity phase based on network configuration

Parameters kwargs (*self.__dict__*) – All arguments are inherited from Sorn attributes

Returns Weight matrices WEI, WEE, WIE and threshold matrices Te, Ti and Initial state vectors X,Y

Return type tuple(array)

ip(*te: numpy.array*, *x: numpy.array*)

Intrinsic Plasiticity mechanism :param te: Threshold vector of excitatory units :type te: array :param x: Excitatory network activity :type x: array

Returns Threshold vector of excitatory units

Return type te (array)

istdp(*wei: numpy.array*, *x: numpy.array*, *y: numpy.array*, *cutoff_weights: list*)

Apply iSTDP rule, which regulates synaptic strength between the pre inhibitory(Xj) and post Excitatory(Xi) synaptic neurons :param wei: Synaptic strengths from inhibitory to excitatory :type wei: array :param x: Excitatory network activity :type x: array :param y: Inhibitory network activity :type y: array :param cutoff_weights: Maximum and minimum weight ranges :type cutoff_weights: list

Returns Synaptic strengths from inhibitory to excitatory

Return type wei (array)

static ss(*wee: numpy.array*)

Synaptic Scaling or Synaptic Normalization :param wee: Weight matrix :type wee: array

Returns Scaled Weight matrix

Return type wee (array)

stdp(*wee: numpy.array*, *x: numpy.array*, *cutoff_weights: list*)

Apply STDP rule : Regulates synaptic strength between the pre(Xj) and post(Xi) synaptic neurons :param wee: Weight matrix :type wee: array :param x: Excitatory network activity :type x: array :param cutoff_weights: Maximum and minimum weight ranges :type cutoff_weights: list

Returns Weight matrix

Return type wee (array)

static structural_plasticity(*wee: numpy.array*)

Add new connection value to the smallest weight between excitatory units randomly :param wee: Weight matrix :type wee: array

Returns Weight matrix

Return type wee (array)

3.1.3 MatrixCollection

The following methods are available via `MatrixCollection`

class `sorn.sorn.`**`MatrixCollection`**(*phase: str*, *state: Optional[dict] = None*)

Collect all matrices initialized and updated during simulation(plasiticity and training phases)

Parameters

- **`phase`** (*`str`*) – Training or Plasticity phase
- **`state`** (*`dict`*) – Network activity, threshold and connection matrices

Returns MatrixCollection instance

`network_activity_t`(*excitatory_net: numpy.array*, *inhibitory_net: numpy.array*, *i: int*)

Network state at current time step

Parameters

- **`excitatory_net`** (*`array`*) – Excitatory network activity
- **`inhibitory_net`** (*`array`*) – Inhibitory network activity
- **`i`** (*`int`*) – Time step

Returns Updated Excitatory and Inhibitory states

Return type tuple(array)

`network_activity_t_1`(*x: numpy.array*, *y: numpy.array*, *i: int*)

Network activity at previous time step

Parameters

- **`x`** (*`array`*) – Excitatory network activity
- **`y`** (*`array`*) – Inhibitory network activity
- **`i`** (*`int`*) – Time step

Returns Previous Excitatory and Inhibitory states

Return type tuple(array)

`threshold_matrix`(*te: numpy.array*, *ti: numpy.array*, *i: int*)

Update threshold matrices

Parameters

- **`te`** (*`array`*) – Excitatory threshold
- **`ti`** (*`array`*) – Inhibitory threshold
- **`i`** (*`int`*) – Time step

Returns Threshold Matrices Te and Ti

Return type tuple(array)

`weight_matrix`(*wee: numpy.array*, *wei: numpy.array*, *wie: numpy.array*, *i: int*)

Update weight matrices

Parameters

- **`wee`** (*`array`*) – Excitatory-Excitatory weight matrix
- **`wei`** (*`array`*) – Inhibitory-Excitatory weight matrix

- **wie** (*array*) – Excitatory-Inhibitory weight matrix
- **i** (*int*) – Time step

Returns Weight Matrices Wee, Wei, Wie

Return type tuple(array)

3.1.4 NetworkState

The following methods are available via `NetworkState`

`class sorn.sorn.NetworkState(`*v_t: numpy.array*`)`

The evolution of network states

Parameters **v_t** (*array*) – External input/stimuli

Returns NetworkState instance

Return type instance(object)

`excitatory_network_state(`*wee: numpy.array*, *wei: numpy.array*, *te: numpy.array*, *x: numpy.array*, *y: numpy.array*, *white_noise_e: numpy.array*`)`

Activity of Excitatory neurons in the network

Parameters

- **wee** (*array*) – Excitatory-Excitatory weight matrix
- **wei** (*array*) – Inhibitory-Excitatory weight matrix
- **te** (*array*) – Excitatory threshold
- **x** (*array*) – Excitatory network activity
- **y** (*array*) – Inhibitory network activity
- **white_noise_e** (*array*) – Gaussian noise

Returns Current Excitatory network activity

Return type x(array)

`incoming_drive(`*weights: numpy.array*, *activity_vector: numpy.array*`)`

Excitatory Post synaptic potential towards neurons in the reservoir in the absence of external input

Parameters

- **weights** (*array*) – Synaptic strengths
- **activity_vector** (*list*) – Acitivity of inhibitory or Excitatory neurons

Returns Excitatory Post synaptic potential towards neurons

Return type incoming(array)

`inhibitory_network_state(`*wie: numpy.array*, *ti: numpy.array*, *y: numpy.array*, *white_noise_i: numpy.array*`)`

Activity of Excitatory neurons in the network

Parameters

- **wee** (*array*) – Excitatory-Excitatory weight matrix
- **wie** (*array*) – Excitatory-Inhibitory weight matrix

- **ti** (*array*) – Inhibitory threshold
- **y** (*array*) – Inhibitory network activity
- **white_noise_i** (*array*) – Gaussian noise

Returns Current Inhibitory network activity

Return type y(array)

recurrent_drive(*wee: numpy.array*, *wei: numpy.array*, *te: numpy.array*, *x: numpy.array*, *y: numpy.array*, *white_noise_e: numpy.array*)

Network state due to recurrent drive received by the each unit at time t+1. Activity of Excitatory neurons without external stimuli

Parameters

- **wee** (*array*) – Excitatory-Excitatory weight matrix
- **wei** (*array*) – Inhibitory-Excitatory weight matrix
- **te** (*array*) – Excitatory threshold
- **x** (*array*) – Excitatory network activity
- **y** (*array*) – Inhibitory network activity
- **white_noise_e** (*array*) – Gaussian noise

Returns Recurrent network state

Return type xt(array)

3.1.5 Simulator

The following methods are available via `Simulator_`

class sorn.sorn.**Simulator_**

Simulate SORN using external input/noise using the fresh or pretrained matrices

Parameters

- **inputs** (*np.array, optional*) – External stimuli. Defaults to None.
- **phase** (*str, optional*) – Plasticity phase. Defaults to "plasticity".
- **matrices** (*dict, optional*) – Network states, connections and threshold matrices. Defaults to None.
- **timesteps** (*int, optional*) – Total number of time steps to simulate the network. Defaults to 1.
- **noise** (*bool, optional*) – If True, noise will be added. Defaults to True.

Returns

Network states, connections and threshold matrices

X_all(array): Excitatory network activity collected during entire simulation steps

Y_all(array): Inhibitory network activity collected during entire simulation steps

R_all(array): Recurrent network activity collected during entire simulation steps

frac_pos_active_conn(list): Number of positive connection strengths in the network at each time step during simulation

Return type last_state(dict)

run(*inputs: Optional[numpy.array] = None*, *phase: str = 'plasticity'*, *state: Optional[dict] = None*, *timesteps: Optional[int] = None*, *noise: bool = True*, *freeze: Optional[list] = None*, *callbacks: list = []*, ***kwargs*)

Simulation/Plasticity phase

Parameters

- **inputs** (*np.array, optional*) – External stimuli. Defaults to None.
- **phase** (*str, optional*) – Plasticity phase. Defaults to "plasticity"
- **state** (*dict, optional*) – Network states, connections and threshold matrices. Defaults to None.
- **timesteps** (*int, optional*) – Total number of time steps to simulate the network. Defaults to 1.
- **noise** (*bool, optional*) – If True, noise will be added. Defaults to True.
- **freeze** (*list, optional*) – List of synaptic plasticity mechanisms which will be turned off during simulation. Defaults to None.
- **callbacks** (*list, optional*) – Requested values from ["ExcitatoryActivation", "InhibitoryActivation", "RecurrentActivation", "WEE", "WEI", "TE", "EEConnectionCounts"] collected and returned from the simulate sorn object.

Returns

Network states, connections and threshold matrices

callback_values(dict): Requexted network parameters and activations

Return type last_state(dict)

update_callback_state(**args*) → None

3.1.6 Trainer

The following methods are available via `Trainer_`

class `sorn.sorn.Trainer_`

Train the network with the fresh or pretrained network matrices and external stimuli

run(*inputs: Optional[numpy.array] = None*, *phase: str = 'training'*, *state: Optional[dict] = None*, *timesteps: Optional[int] = None*, *noise: bool = True*, *freeze: Optional[list] = None*, *callbacks: list = []*, ***kwargs*)

Train the network with the fresh or pretrained network matrices and external stimuli

Args: inputs(np.array, optional): External stimuli. Defaults to None.

phase(str, optional): Training phase. Defaults to "training".

state(dict, optional): Network states, connections and threshold matrices. Defaults to None.

timesteps(int, optional): Total number of time steps to simulate the network. Defaults to 1.

noise(bool, optional): If True, noise will be added. Defaults to True.

freeze(list, optional): List of synaptic plasticity mechanisms which will be turned off during simulation. Defaults to None.

max_workers(int, optional): Maximum workers for multhreading the plasticity steps

Returns

Network states, connections and threshold matrices

X_all(array): Excitatory network activity collected during entire simulation steps

Y_all(array): Inhibitory network activity collected during entire simulation steps

R_all(array): Recurrent network activity collected during entire simulation steps

frac_pos_active_conn(list): Number of positive connection strengths in the network at each time step during simulation

Return type last_state(dict)

`update_callback_state`(**args*) → None

3.2 Utility Functions

3.2.1 Plotting

The following methods are available via `Plotter`

class `sorn.utils.Plotter`

Wrapper class to call plotting methods

static `correlation`(*corr: numpy.array*, *savefig: bool*)

Plot correlation between neurons

Parameters

- **corr** (*array*) – Correlation matrix
- **savefig** (*bool*) – If true will save the plot at the current working directory

Returns Neuron Correlation plot

Return type matplotlib.pyplot

static `hamming_distance`(*hamming_dist: list*, *savefig: bool*)

Hamming distance between true netorks states and perturbed network states

Parameters

- **hamming_dist** (*list*) – Hamming distance values
- **savefig** (*bool*) – If True, save the fig at current working directory

Returns Hamming distance between true and perturbed network states

Return type matplotlib.pyplot

static `hist_firing_rate_network`(*spike_train: numpy.array*, *bin_size: int*, *savefig: bool*)

Plot the histogram of firing rate (total number of neurons spike at each time step)

Parameters

- **spike_train** (*array*) – Array of spike trains
- **bin_size** (*int*) – Histogram bin size
- **savefig** (*bool*) – If True, plot will be saved in the cwd

Returns plot object

static hist_incoming_conn(*weights: numpy.array*, *bin_size: int*, *histtype: str*, *savefig: bool*)

Plot the histogram of number of presynaptic connections per neuron

Parameters

- **weights** (*array*) – Connection weights
- **bin_size** (*int*) – Histogram bin size
- **histtype** (*str*) – Same as histtype matplotlib
- **savefig** (*bool*) – If True plot will be saved as png file in the cwd

Returns plot object

Return type plot (matplotlib.pyplot)

static hist_outgoing_conn(*weights: numpy.array*, *bin_size: int*, *histtype: str*, *savefig: bool*)

Plot the histogram of number of incoming connections per neuron

Parameters

- **weights** (*array*) – Connection weights
- **bin_size** (*int*) – Histogram bin size
- **histtype** (*str*) – Same as histtype matplotlib
- **savefig** (*bool*) – If True plot will be saved as png file in the cwd

Returns plot object

static isi_exponential_fit(*spike_train: numpy.array*, *neuron: int*, *bin_size: int*, *savefig: bool*)

Plot Exponential fit on the inter-spike intervals during training or simulation phase

Parameters

- **spike_train** (*array*) – Array of spike trains
- **neuron** (*int*) – Target neuron
- **bin_size** (*int*) – Spike train will be splitted into bins of size bin_size
- **savefig** (*bool*) – If True, plot will be saved in the cwd

Returns plot object

static linear_lognormal_fit(*weights: numpy.array*, *num_points: int*, *savefig: bool*)

Lognormal curve fit on connection weight distribution

Parameters

- **weights** (*array*) – Connection weights
- **num_points** (*int*) – Number of points to be plotted in the x axis
- **savefig** (*bool*) – If True, plot will be saved in the cwd

Returns plot object

static network_connection_dynamics(*connection_counts: numpy.array*, *savefig: bool*)

Plot number of positive connection in the excitatory pool

Parameters

- **connection_counts** (*array*) –
- **savefig** (*bool*) –

Returns plot object

static **plot_network**(*corr: numpy.array*, *corr_thres: float*, *fig_name: Optional[str] = None*)

Network x graphical visualization of the network using the correlation matrix

Parameters

- **corr** (*array*) – Correlation between neurons
- **corr_thres** (*array*) – Threshold to prune the connection. Smaller the threshold, higher the density of connections
- **fig_name** (*array, optional*) – Name of the figure. Defaults to None.

Returns Plot instance

Return type matplotlib.pyplot

static **raster_plot**(*spike_train: numpy.array*, *savefig: bool*)

Raster plot of spike trains

Parameters

- **spike_train** (*array*) – Array of spike trains
- **with_firing_rates** (*bool*) – If True, firing rate of the network will be plotted
- **savefig** (*bool*) – If True, plot will be saved in the cwd

Returns plot object

static **scatter_plot**(*spike_train: numpy.array*, *savefig: bool*)

Scatter plot of spike trains

Parameters

- **spike_train** (*list*) – Array of spike trains
- **with_firing_rates** (*bool*) – If True, firing rate of the network will be plotted
- **savefig** (*bool*) – If True, plot will be saved in the cwd

Returns plot object

static **weight_distribution**(*weights: numpy.array*, *bin_size: int*, *savefig: bool*)

Plot the distribution of synaptic weights

Parameters

- **weights** (*array*) – Connection weights
- **bin_size** (*int*) – Spike train will be splited into bins of size bin_size
- **savefig** (*bool*) – If True, plot will be saved in the cwd

Returns plot object

3.2.2 Statistics and Analysis

The following methods are available via `Statistics`

class `sorn.utils.`**`Statistics`**

Wrapper class for statistical analysis methods

static **`autocorr`**(*firing_rates: list*, *t: int = 2*)

Score interpretation - scores near 1 imply a smoothly varying series - scores near 0 imply that there's no overall linear relationship between a data point and the following one (that is, plot(x[-length(x)],x[-1]) won't give a scatter plot with any apparent linearity)

- scores near -1 suggest that the series is jagged in a particular way: if one point is above the mean, the next is likely to be below the mean by about the same amount, and vice versa.

Parameters

- **firing_rates** (*list*) – Firing rates of the network
- **t** (*int, optional*) – Window size. Defaults to 2.

Returns Autocorrelation between neurons given their firing rates

Return type array

static **`avg_corr_coeff`**(*spike_train: numpy.array*)

Measure Average Pearson correlation coeffecient between neurons

Parameters **spike_train** (*array*) – Neural activity

Returns Average correlation coeffecient

Return type array

static **`fanofactor`**(*spike_train: numpy.array*, *neuron: int*, *window_size: int*)

Investigate whether neuronal spike generation is a poisson process

Parameters

- **spike_train** (*np.array*) – Spike train of neurons in the reservoir
- **neuron** (*int*) – Target neuron in the pool
- **window_size** (*int*) – Sliding window size for time step ranges to be considered for measuring the fanofactor

Returns Fano factor of the neuron spike train

Return type float

static **`firing_rate_network`**(*spike_train: numpy.array*)

Calculate number of neurons spikes at each time step.Firing rate of the network

Parameters **spike_train** (*array*) – Array of spike trains

Returns firing_rate

Return type int

static **`firing_rate_neuron`**(*spike_train: numpy.array*, *neuron: int*, *bin_size: int*)

Measure spike rate of given neuron during given time window

Parameters

- **spike_train** (*array*) – Array of spike trains

- **neuron** (*int*) – Target neuron in the reservoir
- **bin_size** (*int*) – Divide the spike trains into bins of size bin_size

Returns firing_rate

Return type int

`static hamming_distance`(*actual_spike_train: numpy.array*, *perturbed_spike_train: numpy.array*)

Hamming distance between true netorks states and perturbed network states

Parameters

- **actual_spike_train** (*np.array*) – True network's states
- **perturbed_spike_train** (*np.array*) – Perturbated network's states

Returns Hamming distance between true and perturbed network states

Return type float

`static scale_dependent_smoothness_measure`(*firing_rates: list*)

Smoothem the firing rate depend on its scale. Smaller values corresponds to smoother series

Parameters **firing_rates** (*list*) – List of number of active neurons per time step

Returns Float value signifies the smoothness of the semantic changes in firing rates

Return type sd_diff (list)

`static scale_independent_smoothness_measure`(*firing_rates: list*)

Smoothem the firing rate independent of its scale. Smaller values corresponds to smoother series

Parameters **firing_rates** (*list*) – List of number of active neurons per time step

Returns Float value signifies the smoothness of the semantic changes in firing rates

Return type coeff_var (list)

`static spike_source_entropy`(*spike_train: numpy.array*, *num_neurons: int*)

Measure the uncertainty about the origin of spike from the network using entropy

Parameters

- **spike_train** (*np.array*) – Spike train of neurons
- **num_neurons** (*int*) – Number of neurons in the reservoir

Returns Spike source entropy of the network

Return type int

`static spike_time_intervals`(*spike_train*)

Generate spike time intervals spike_trains

Parameters **spike_train** (*array*) – Network activity

Returns Inter spike intervals for each neuron in the reservoir

Return type list

`static spike_times`(*spike_train: numpy.array*)

Get the time instants at which neuron spikes

Parameters **spike_train** (*array*) – Spike trains of neurons

Returns Spike time of each neurons in the pool

Return type (array)

CHAPTER FOUR

CONTRIBUTIONS

If you wish to contribute, please

1. Fork the github repo as:

```
git clone git@github.com:your-user-name/sorn.git sorn-yourname
cd sorn-yourname
git remote add upstream git://github.com/saran-nns/sorn.git
```

2. Create a branch as:

```
git checkout -b your_branch_name
```

3. Before pull request, please retrieve the changes from the sorn *master-branch* as:

```
git fetch master
git rebase master
```

and the changes can be discussed there.

If you find a bug in the code or errors in the documentation, please open a new issue in the Github repository and report the bug or the error. Please provide sufficient information for the bug to be reproduced.

CHAPTER FIVE

CITE PACKAGE

Please cite the package as:

```
@software{saranraj_nambusubramaniyan_2020_4184103,
author       = {Saranraj Nambusubramaniyan},
title        = {Saran-nns/sorn: Stable alpha release},
month        = nov,
year         = 2020,
publisher    = {Zenodo},
version      = {v0.3.1},
doi          = {10.5281/zenodo.4184103},
url          = {https://doi.org/10.5281/zenodo.4184103}
}
```

CHAPTER

SIX

CITE PAPER

Please cite my thesis as:

```
Saranraj Nambusubramaniyan(2019): Prospects of Biologically Plausible Artificial Brain␣
↪Circuits Solving General Intelligence Tasks at the Imminence of Chaos DOI: 10.13140/RG.
↪2.2.25393.81762
```

CHAPTER SEVEN

MIT LICENSE

CHAPTER EIGHT

CONTACT

Question? Please contact *saran_nns@hotmail.com*

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search