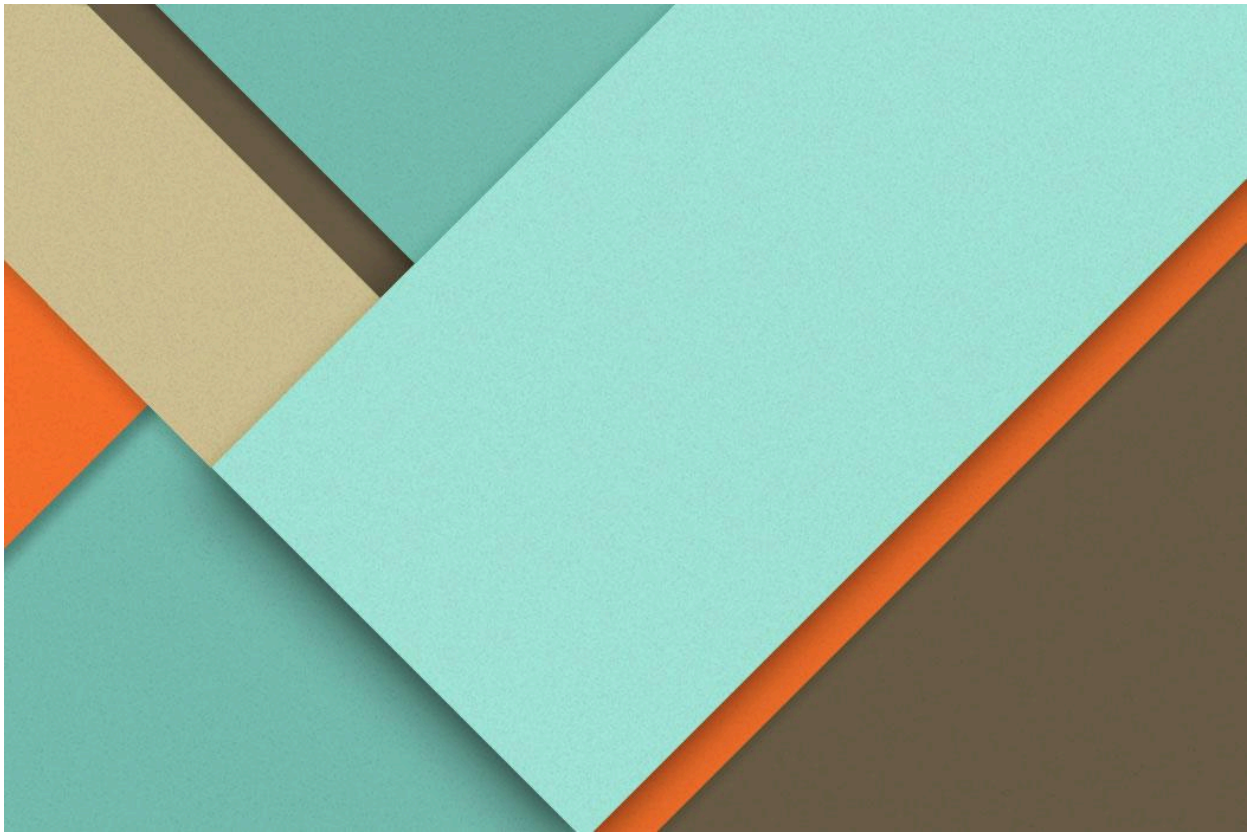


Project Report on STDP and RSTDP Models

—

Katayoon Kobraei
Shahid Beheshti University of Tehran

Overview

In this project, we designed two models by building on the previous exercises, including the examination of the LIF model and the study of inter-neuronal populations. This report explain the assignment, focusing on the learning process in neurons using Spike-Timing Dependent Plasticity (STDP). STDP is a critical mechanism for synaptic modification based on the timing of spikes between connected neurons, making it essential for understanding how learning occurs in neural circuits. The objective of this assignment is to implement and analyze the effects of STDP in various neuronal networks, exploring how synaptic weights evolve based on different input stimuli and learning strategies.

Goals

The assignment consists of two main parts:

1. **STDP Learning**: A network of three interconnected neurons is studied under different input currents, such as constant and sinusoidal currents. The goal is to observe how the synaptic weights between these neurons change based on the STDP learning rule and to analyze the effect of varying input types on the network.
2. **Reward-Based STDP Learning**: A more advanced task involves simulating a simple neural network with input and output layers and training it using reward-based STDP. The performance of this network is evaluated by adjusting synaptic weights to achieve correct spike outputs in response to specific input patterns. The dataset used includes both training and testing sections, with predefined input currents and expected output spikes.

Through these experiments, this assignment aims to deepen the understanding of neuronal learning mechanisms and how synaptic plasticity adapts to different stimuli and reinforcement signals.

Part 1

In the first section, we design the STDP neuron. To do this, we first need a simple neuron class that includes standard features such as current, resistance, etc. Then, we implement related functions like defining potential energy values, frequencies, and plotting functions for this class.

```
class Neuron:
    def __init__(self,i_func = 0, i = 5, time_interval = 100, dt = 0.1, u_rest = 0, R = 1, C = 10, threshold = 1,reset = True
                 ,save_name="none", u_spike = 20, u_reset = 0, tau = 0):
        self.i = i
        self.time_interval = time_interval
        self.dt = dt
        self.u_rest = u_rest
        self.R = R
        self.C = C
```

```
def process(self, current_function, timespan, dt, reset=True):
    if reset:
        self.to_rest()
    size = math.ceil(timespan / dt)
    U = np.zeros(shape=(size, 2))
    spikes = []
    time = 0
    for index in range(len(U)):
        U[index, 1] = self.u
        U[index, 0] = time
        if self.u > self.threshold:
            spikes.append(time)
            self.reset()
        du = dt * (-1 * (self.u - self.u_rest) + 1e-3 * self.R * current_function(time)) / self.tau
        self.u += du
        time += dt
    return {'voltage': U, 'spikes': spikes}

def frequency(self, current_range, timespan, dt):
    data = np.zeros(shape=(len(current_range), 2))
    for index in range(len(current_range)):
        self.to_rest()
        Func = lambda x: current_range[index]
        result = self.process(Func, timespan=timespan, dt=dt)
        result = result['spikes']
        data[index, 0] = current_range[index]
        if len(result) == 0:
            data[index, 1] = 0
        elif len(result) == 1:
            data[index, 1] = 1 / timespan
        else:
            data[index, 1] = (len(result) - 1) / (result[-1] - result[0])
    return data
```

In the next class, we implement the connection between neurons and neuronal populations, similar to the previous exercises. This class is defined with additional features for neuronal populations, such as specifying whether the population is excitatory or inhibitory, and the neurons within this population, which are objects from the previous neuron class.

```
class Population:
    def __init__(self, population_type, neurons, time_course, j=20):
        self.population_activity = []
        self.neurons = neurons
        self.connections = np.zeros((len(neurons), len(neurons)))
        self.j = j
        self.connection_type = self.fully_connection
        set_connection = self.connection_type
        self.population_type = population_type
        set_connection()
        self.time_course = time_course
        self.connection_history = deepcopy(self.connections.ravel())
```

Next, we implement the required functions for a neuronal population:

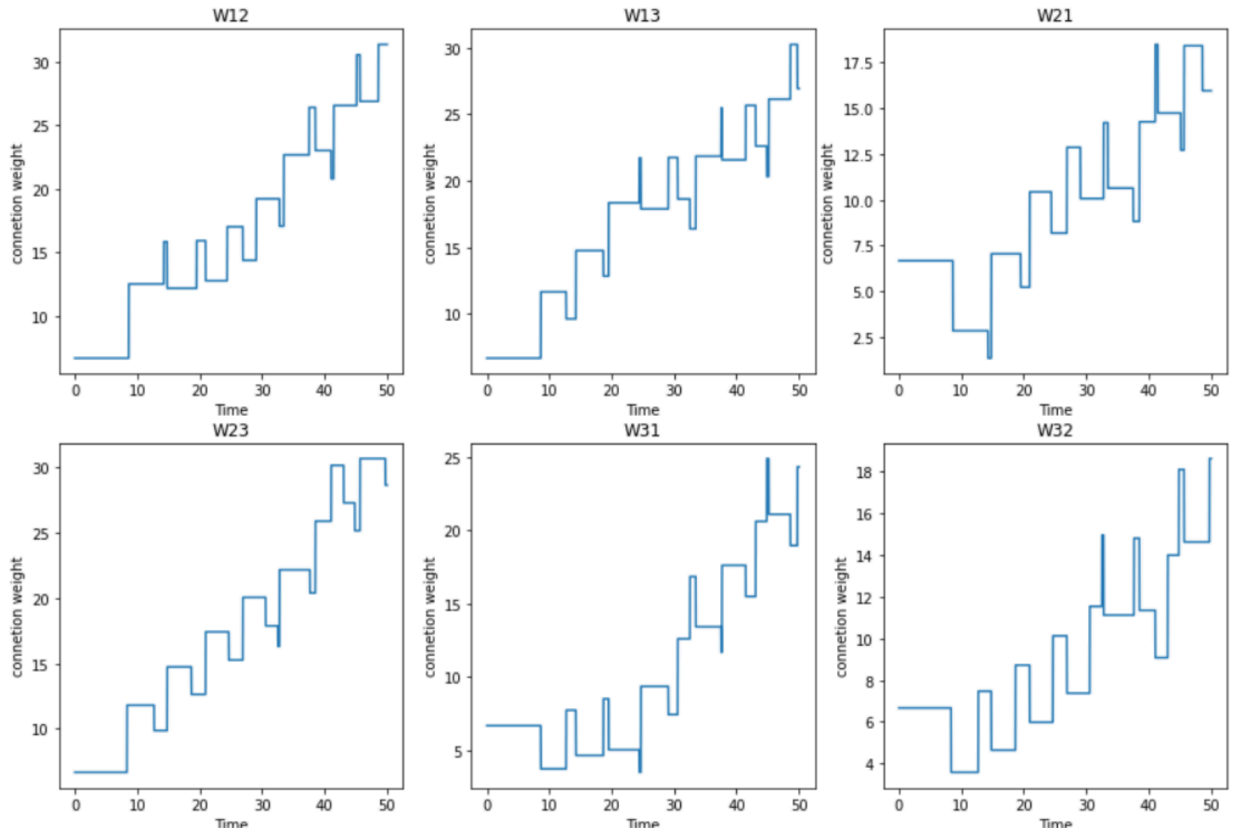
Note: The implemented population is a Connective_FULL population.

```
def fully_connection(self):
    self.connections = np.ones_like(self.connections) * (self.j / len(self.neurons))
    if self.population_type == 'inhibitory':
        self.connections = -1 * self.connections

def activity_history(self, time, dt, threshold):
    activity_list = np.zeros((len(self.neurons), 1))
    for idx in range(len(self.neurons)):
        activity_list[idx, 0] = self.activity_history_single(idx, time, dt, threshold)
    return activity_list

def activity_history_single(self, idx, time, dt, threshold):
    neuron = self.neurons[idx]
    S = 0
    activity = 0
    while self.time_course(S) > threshold:
        if (time - S) in neuron.spikes:
            activity += self.time_course(S)
        S += dt
    return activity

def single_step(self, input_current, self_activity, time, dt, time_course_threshold):
    inputs = self.connections.dot(self_activity)
    for i, neuron in enumerate(self.neuron_list):
        neuron.single_step(input_current + inputs[i, 0], time, dt)
    activity = self.calculate_activity_history(time + dt, dt, time_course_threshold)
    return activity

def reset(self, reset_connection=False):
    self.population_activity = []
    if reset_connection:
        set_connection = self.connection_type
        set_connection()
    for neuron in self.neurons:
        neuron.clear_history()
```

Examples

First, we need to implement three neuron models and pass them in a list to the neuronal population class. Then, we input three different currents into the STDP class. The resulting graphs will be as follows:

```
s = STDP("5000", "4000 * (math.sin(x) + 0.9)", "5000 * (math.cos(x) + 0.9)")
s.weight_plotting()
```



Based on the graphs, we can observe that when the postsynaptic neuron fires, the synaptic weights decrease, and conversely, when the presynaptic neuron fires, the weights increase.

Parrt 2

In this section, we first need to design a class for the SNN (Spiking Neural Network) model, where we implement the learning algorithm. Then, we load the desired dataset from a source.

	test	Unnamed: 1	Unnamed: 2	Unnamed: 3	Unnamed: 4	Unnamed: 5	Unnamed: 6	Unnamed: 7	Unnamed: 8	Unnamed: 9	Unnamed: 10
0	inpput_neuron_number	train_1	train_2	train_3	train_4	train_5	train_6	train_7	train_8	train_9	train_10
1	1.0	1.0	2.0	3.0	2.0	1.0	1.0	0.0	0.0	0.0	2.0
2	2.0	1.0	1.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	2.0
3	3.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0
4	4.0	0.0	0.0	0.0	0.0	0.0	0.0	3.0	1.0	0.0	1.0
5	5.0	0.0	0.0	0.0	0.0	1.0	2.0	0.0	1.0	1.0	2.0
6	output_neuron_number	1.0	1.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0
7	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
8	test	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
9	inpput_neuron_number	train_1	train_2	train_3	train_4	train_5	train_6	train_7	train_8	train_9	train_10
10	1.0	1.0	0.0	0.0	2.0	1.0	1.0	3.0	0.0	1.0	2.0
11	2.0	0.0	1.0	2.0	2.0	2.0	0.0	1.0	0.0	2.0	2.0
12	3.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	2.0	0.0	2.0
13	4.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	2.0	1.0	3.0
14	5.0	0.0	0.0	0.0	0.0	0.0	3.0	3.0	2.0	1.0	3.0
15	output_neuron_number	1.0	1.0	1.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0

As observed, the data needs to be multiplied by 10,000 for input into the algorithm.

When we run the model, we can see that the model has learned to a good extent.

```
accuracy of SNN on test data:  70.0 %
accuracy of SNN on train data:  80.0 %
```

Finally, the graphs for this part will be as follows: