Lab manual
Synaptic plasticity on SpiNNaker with PyNN

1 – Introduction

This portion of the manual introduces the methodology to instantiate and perform a spiking neural network simulation on SpiNNaker using PyNN.

2 – Installation of the plugin

The main software package to be used for this lab is sPyNNaker. However a few models are additional to the basic package and require the sPyNNakerExtraModelsPlugin which can be installed using the command

```
sudo pip install sPyNNakerExtraModelsPlugin
```

for a system-wide installation, or for a different method (e.g. virtualenv), please refer to sPyNNaker module installation instructions, adapting the commands to the sPyNNakerExtraModelsPlugin module.

3 – Spike Timing-Dependent Plasticity (STDP)

This class of synaptic learning is induced by tight temporal correlation between a pre-synaptic and a post-synaptic spike event. It is a temporally asymmetric form of Hebbian learning, and it is believed to be at the basis of learning and information storage in the human brain.

On the basis of this rule, in the case where a pre-synaptic spike is tightly followed by a post-synaptic spike, then there is a causal relationship between the two events, and the synapse which carried the pre-synaptic event is potentiated.

On the other hand, in the case the post-synaptic spike is emitted shortly before the post-synaptic spike is emitted, an anti-causal relation is present between those spikes, and the synapse which carried the pre-synaptic event is depressed.

This learning rule may be summarised with the help of the following figure [Sjöström and Gerstner (2010), Scholarpedia]:

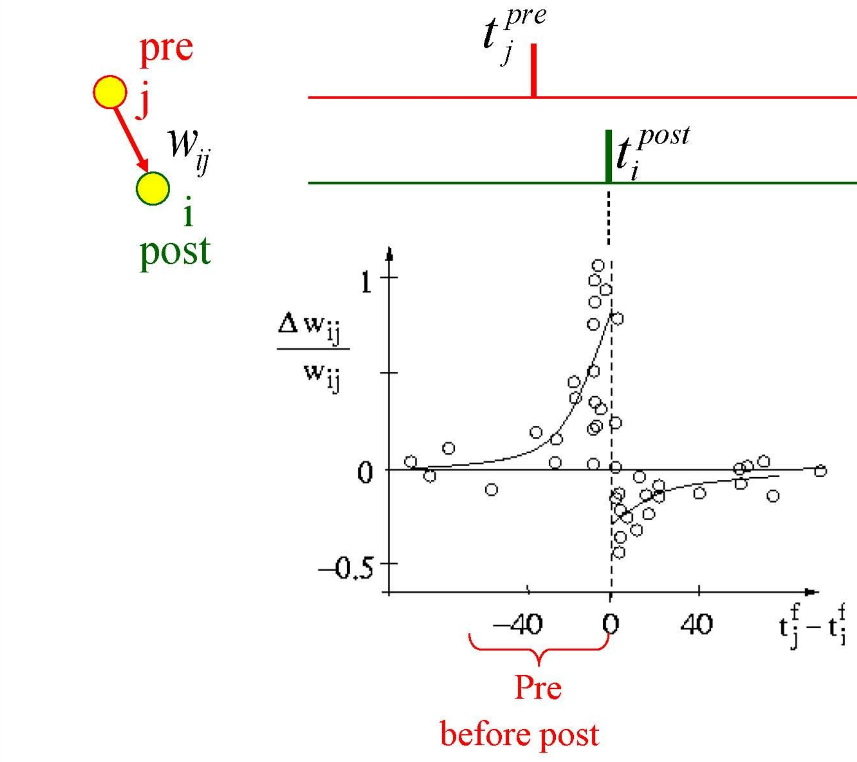


Illustration 1: Description of the STDP learning rule

The image highlights that the amount of potentiation or depression depends on the relative timing of the pre- and post-synaptic spikes using the double-exponential-like function. If the time of the post-synaptic spike is centred on the 0 of the function, then the pre-synaptic spike time highlights how much potentiation/depression the synapse undergoes.

However, the function is not univocal, as different experiments have highlighted different behaviours depending on the conditions (e.g. [Graupner and Brunel (2012), PNAS]). Other authors have also suggested a correlation between triplets and quadruplets of pre- and post-synaptic spikes to trigger adequately synaptic potentiation or depression.

4 – Plasticity in PyNN

In PyNN the synaptic plasticity can be instantiated using the `SynapseDynamics` class. This class identifies two main types of dynamics: slow and fast. SpiNNaker only implements slow STDP rules, which in PyNN are identified by the `STDPMechanism` class.

The constructor for the `SynapseDynamics` class is:

```
SynapseDynamics(fast, slow)
```

Where each of the parameter is optional, and therefore it may describe either a fast rule (short-term plasticity – STP), slow rule (spike timing-dependent plasticity – STDP) or both.

As described earlier, the STDP rule depends on the relative timing between pre- and post-synaptic spikes, and the final synaptic weight depends on the weight before the rule is applied. These dependencies actually reflect the way the class `STDPMechanism` is instantiated in PyNN:

```
STDPMechanism(timing_dependence, weight_dependence,
              voltage_dependence)
```

PyNN also allows for a voltage dependence, however, this is not currently used in any of the rules implemented on SpiNNaker, therefore will not be included in later descriptions).

Each of these dependencies is optional, so that any combination of them can be used by learning rules. If we consider the general STDP rule, each portion of the rule can be specified as:

```
timing_dependence = SpikePairRule(tau_plus, tau_minus)
```

The two parameters of this class identify the exponential decay rate of the STDP function, as described in figure 1 earlier in this manual.

The second of these dependencies is on weight, as the final synaptic weight depends on the weight of the synapse before the learning rule is applied. This dependency is specified based on an additive or a multiplicative weight dependence through two different classes, as specified below:

```
weight_rule = AdditiveWeightDependence(
                                    w_min, w_max, A_plus, A_minus)

weight_rule = MultiplicativeWeightDependence(
                                    w_min, w_max, A_plus, A_minus)
```

The parameters of these class can be described using the two figures below, where the vertical axis represents the synaptic weight evolution, and the horizontal axis represents time:

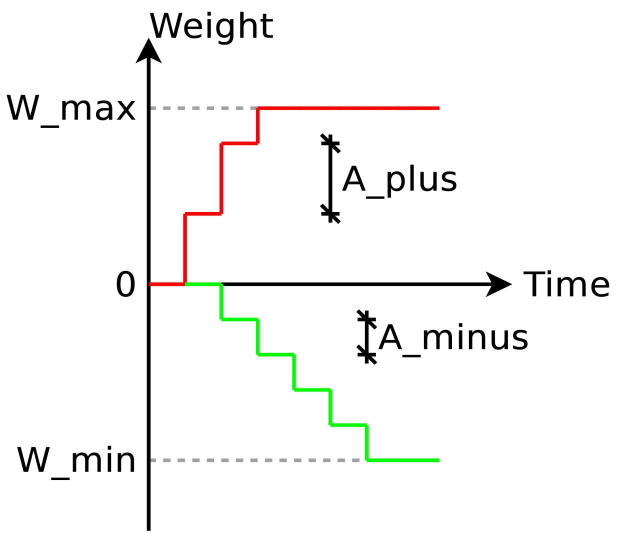


Illustration 2: Additive weight dependence parameters

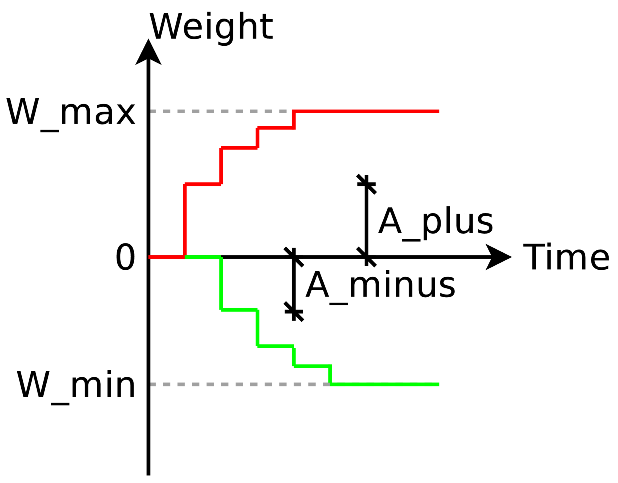


Illustration 3: Multiplicative weight dependence parameters

W_min and W_max always represent saturation values, which the synaptic weight cannot exceed.

Combining all these class instantiations in a simple example script to define a learning rule:

```
time_rule = SpikePairRule(tau_plus=1, tau_minus=1)

weight_rule = AdditiveWeightDependence(
                      w_min=0.0, w_max=2, A_plus=0.5, A_minus=0.5)


stdp_model = STDPMechanism(
   timing_dependence = time_rule, weight_dependence = weight_rule)


syn_dyn = SynapseDynamics(slow = stdp_model)


proj = Projection(
  pop_src, pop_dst, p.AllToAllConnector(weights, delays), syn_dyn)
```

The last command instantiates a projection between a source population (`pop_src`) and a destination population (`pop_dst`).

It is possible to retrieve synaptic parameters (weights and delays) at the end of a simulation, using standard PyNN API function calls:

```
weights = proj.getWeights([format='list' | 'array'])

delays = proj.getDelays([format='list' | 'array'])
```

The parameter “format” in this function call is optional, and it identifies the format of the weight or delay structure that is returned. Weights or delays returned as “list” identify each single active synapse, but not pre- and post-synaptic neuron IDs. This may be useful to process weights or delays considering only active synapses, rather than all the possible combinations. For specific analysis, in which pre- and post-synaptic neuron ID is required, the second format, “array”, specifies that weights and delays are returned as matrix, where each row identifies a pre-synaptic neuron and each column identifies a post-synaptic neuron. In case there is no synapse for specific pre- and post-synaptic neuron combination(s), a value of NaN is returned in the appropriate matrix location(s).

5 – Tasks with synaptic plasticity

In the following example, often a synapse is required to have enough weight to make the post-synaptic neuron fire exactly once.

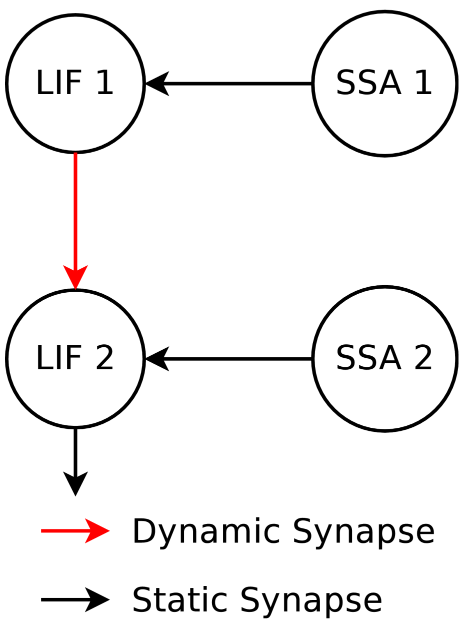
To help the development of such condition, an example of such condition is described here. Using an LIF neuron (`IF_curr_exp`) with the following parameters requires a synaptic weight of 2.0 to generate an output spike for each incoming spike:

```
cell_params_lif = {'cm'         : 0.25, # nF
                   'i_offset'   : 0.0,
                   'tau_m'      : 20.0,
                   'tau_refrac': 2.0,
                   'tau_syn_E' : 5.0,
                   'tau_syn_I' : 5.0,
                   'v_reset'    : -70.0,
                   'v_rest'     : -65.0,
                   'v_thresh'   : -50.0
                   }
```

Other combinations of parameters and synaptic weights are possible, and can be easily tested by each participant.

Task 1 – Simple supervised learning [Easy]

Write a network with two LIF neurons connected with a plastic synapse with initial weight 0. A third SpikeSourceArray neuron excites the first LIF neuron which spikes. A fourth SpikeSourceArray neuron excites the second LIF neuron which spikes (see image below). This last neuron is called the “teaching neuron”, and has a synaptic weight so high that the post-synaptic neuron fires exactly once for every incoming spike. If the SpikeSourceArray neurons are timed correctly, the plastic synapse should show potentiation.



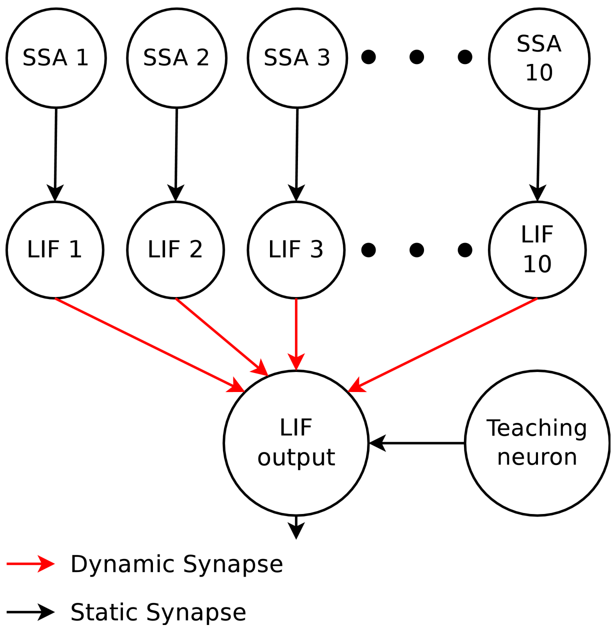
If the network is described in such way that neuron LIF 1 spikes, neuron LIF 2 receives the spike and then fires, the synapse between LIF 1 and LIF 2 should result potentiated. This can be tested checking that LIF 2 is able to fire as effect of excitation coming form the LIF 1 neuron. Synaptic weights can be retrieved and displayed to highlight modification during the simulation.

Task 2 – Simple supervised learning [Easy]

In the network described in task modify the initial conditions so that the initial weight of the synapse is relatively high, but lower than the weight required by LIF 2 to fire. Then the timing of the spikes needs to be changed in such a wat that neuron LIF 2 fires and after receives a spike from neuron LIF 1. At the end of the simulation this task will show depression of the plastic synapse between LIF 1 and LIF 2.

Task 3 – Supervised learning with multiple neurons [Moderate]

This task is a combination of the two previous tasks to show in the same network both synaptic potentiation and depression following supervised learning. Define a set of 10 SpikeSourceArray input neurons which project spikes to a population of 10 LIF neurons with a synaptic weight enough high so that each incoming spike generates an output spike. The 10 LIF neurons project spikes to a single output LIF neuron with plastic synapses and initial weights se to half the weight required to spike. The output LIF neuron has also a teaching neuron attached to it. The network should look like the one in the diagram below:



The spike times for the input and teaching neurons can be set in such a way that the the input neurons fire in a sequence, and the teaching neuron fires at the same time as the middle neuron of the input layer. With the appropriate learning parameters, the synaptic weight for the early-spiking neurons is getting depressed, while the synaptic weights of the neurons after the middle one are getting potentiated. The final weights should reflect the shape of the STDP curve.