Using neuromodulated STDP on SpiNNaker

This document describes how to run models where the synaptic weights can be neuromodulated based upon “reward” and “punishment” values. This work was originally undertaken by Mantas Mikaitis [2] as part of his PhD and has recently been updated and modified to become part of the “master” sPyNNaker [3] toolchain. This implementation is not currently part of the “default” PyNN implementation but as with a number of projects within sPyNNaker, is designed to work with as little modification to how a user would interact with PyNN as possible.

Contents

1. Getting Started

In order to use neuromodulation currently (March 2022) you need to use the git “master” repositories. This can either be done by using Jupyter with the “sPyNNakerGit” kernel, or by installing the tools either in developer mode or from the command line.

Once you are installed, make sure you are up-to-date with the latest changes and ensure that all C and Java code is built.

To illustrate how to make a PyNN script including neurmodulated synaptic weights, consider the following example, taken from

```
PyNN8Examples/examples/stdp_neuromodulation_test.py
```

In this case we are considering a simple weight change based on a “reward” of a single neuron, based upon Izhikevich’s paper from 2007 [1]. Firstly we set parameters for this setup:

```
import spynnaker8 as sim

timestep = 1.0
duration = 3000

# Main parameters from Izhikevich 2007 STDP paper
t_pre = [1500, 2400]    # Pre-synaptic neuron times
t_post = [1502]         # Post-synaptic neuron stimuli time
t_dopamine = [1600]     # Dopaminergic neuron spike times
tau_c = 1000            # Eligibility trace decay time constant.
tau_d = 200             # Dopamine trace decay time constant.
```

```
DA_concentration = 0.1  # Dopamine trace step increase size

# Initial weight
rewarded_syn_weight = 0.0

cell_params = {
      'cm': 0.3,
      'i_offset': 0.0,
      'tau_m': 10.0,
      'tau_refrac': 4.0,
      'tau_syn_E': 1.0,
      'tau_syn_I': 1.0,
      'v_reset': -70.0,
      'v_rest': -65.0,
      'v_thresh': -55.4}
```

Next we call the setup function, and then build a network of stimulus, reward, pre- and post-populations.

```
sim.setup(timestep=timestep)

pre_pop = sim.Population(1, sim.SpikeSourceArray, {
      'spike_times': t_pre})

# Create a population of dopaminergic neurons for reward
reward_pop = sim.Population(1, sim.SpikeSourceArray, {
      'spike_times': t_dopamine}, label='reward')

# Stimulus for post synaptic population
post_stim = sim.Population(1, sim.SpikeSourceArray, {
      'spike_times': t_post})

# Create post synaptic population which will be modulated by DA
concentration.
post_pop = sim.Population(
      1, sim.IF_curr_exp, cell_params,
      label='post1')
```

We connect the stimulus to the post-population:

```
# Stimulate post-synaptic neuron
sim.Projection(
      post_stim, post_pop,
      sim.AllToAllConnector(),
```

```
        synapse_type=sim.StaticSynapse(weight=6),
        receptor_type='excitatory')
```

Next we build a simple STDP model with a timing pair rule and additive weight dependence:

```
# Create STDP dynamics
synapse_dynamics = sim.STDPMechanism(
        timing_dependence=sim.SpikePairRule(
        tau_plus=10, tau_minus=12,
        A_plus=1, A_minus=1),
        weight_dependence=sim.AdditiveWeightDependence(
        w_min=0, w_max=20),
        weight=0.0)
```

And then add this as a projection between the pre- and post-populations:

```
# Create a plastic connection between pre and post neurons
plastic_projection = sim.Projection(
        pre_pop, post_pop,
        sim.AllToAllConnector(),
        synapse_type=synapse_dynamics,
        receptor_type='excitatory', label='Pre-post projection')
```

And finally we create a projection between the reward and post population to tell it that synapses connecting into it are to be neuromodulated:

```
# Create dopaminergic connection
reward_projection = sim.Projection(
        reward_pop, post_pop,
        sim.AllToAllConnector(),
        synapse_type=sim.extra_models.Neuromodulation(
        tau_c=1000, tau_d=200, weight=DA_concentration, w_max=20.0),
        receptor_type='reward', label='reward synapses')
```

Note the order here; the STDP connection is defined first, then neuromodulation is added afterwards; this ordering is critical. The network is now ready so we can run the example, print out the final weight, and end:

```
sim.run(duration)

# End simulation on SpiNNaker
print("Final weight: " + repr(plastic_projection.get('weight',
'list')))
```

```
sim.end()
```

This will give a weight value on SpiNNaker of 10.0654296875, comparable to the value calculated using the equations in [1] of 10.0552710.

This script worked because the binary code required to run it is built in the standard Makefile build. You may find that your particular combination of neuron model and neuromodulated STDP with certain timing or weight rules is not built by default; if this is the case, to build the binary you require, copy the directory and Makefile contained in it at

```
neural_modelling/makefiles/neuron/IF_curr_exp_stdp_izhikevich_neuromodulation_pair_additive
```

into another directory in

```
neural_modelling/makefiles/neuron
```

Edit this new Makefile to use the components you require, and then make sure that this combination is built by adding it to the Makefile in

```
neural_modelling/makefiles/neuron/Makefile
```

and then either running “make” inside this directory, or running the automatic_make.sh script in the SupportScripts repository. You can check that your binary is built at the end of this procedure by having a look for it in

```
sPyNNaker/spynnaker/pyNN/model_binaries
```

2. Further Examples

A further neuromodulated STDP example can be found at

```
PyNN8Examples/examples/stdp_neuromodulated_example.py
```

In this example, we take 10 populations of 5 stimuli neurons and connect to each 10 post-synaptic populations of 5 neurons. The spiking of stimuli causes some spikes in post-synaptic neurons initially. We then inject reward signals from dopaminergic neurons periodically to reinforce synapses that are active. This is followed by increased weights of some synapses and thus increased response to the stimuli. We then proceed to inject punishment signals from dopaminergic neurons which causes an inverse effect to reduce response of post-synaptic neurons to the same stimuli.

The STDP rule is the same as in the example in section 1; the remainder of the script is simply creating the network of the multiple pre- and post-populations described above, and then

plotting the spike raster across the multiple post-populations alongside an indication of where the rewards and punishments take place. See figure 1.

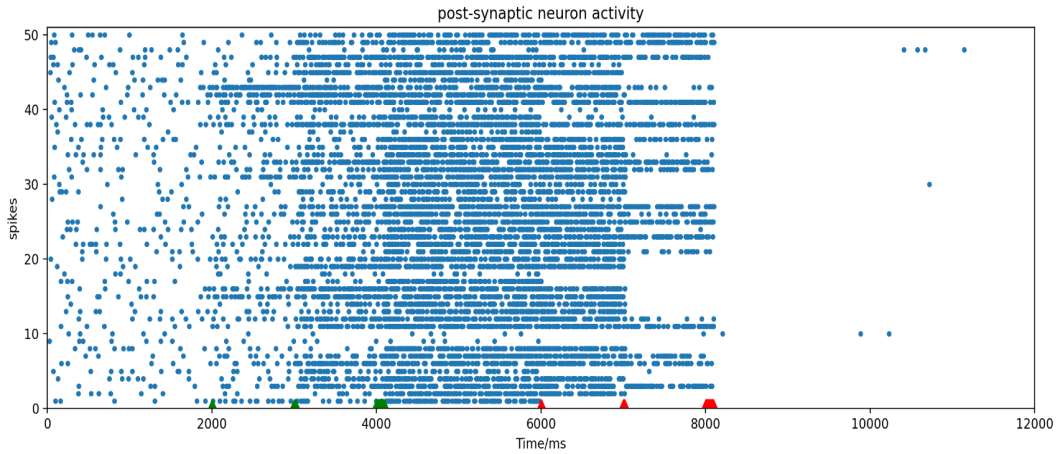

Figure 1: post-synaptic neuron activity across multiple populations affected by rewards (green) followed by punishments (red).

More complicated code binaries are built by default; in particular, it is possible to combine both neuromodulated STDP and structural plasticity (see [4] for details of this implementation). This is shown best in the example in

`PyNN8Examples/examples/split_examples/structural_plasticity_with_stdp_neuromodulated.py`

In this case the synapse dynamics are specified in the following way:

```
# Create synapse dynamics with neuromodulated STDP and structural plasticity
# Structurally plastic connection between pre_pop and post_pop
partner_selection_last_neuron = sim.RandomSelection()
formation_distance = sim.DistanceDependentFormation(
      grid=[np.sqrt(n_neurons), np.sqrt(n_neurons)],  #spatial org of neurons
      sigma_form_forward=0.5  # spread of feed-forward connections
)
elimination_weight = sim.RandomByWeightElimination(
      prob_elim_potentiated=0.2,  # no eliminations for potentiated synapses
      prob_elim_depressed=0.2,  # no elimination for depressed synapses
      threshold=plastic_weights  # Use same weight as initial weight
)

synapse_dynamics = sim.StructuralMechanismSTDP(
      # Partner selection, formation and elimination rules from above
      partner_selection_last_neuron, formation_distance, elimination_weight,
```

```
        # Use this weight when creating a new synapse
        initial_weight=plastic_weights,
        # Use this weight for synapses at start of simulation
        weight=plastic_weights,
        # Use this delay when creating a new synapse
        initial_delay=10,
        # Use this delay for synapses at the start of simulation
        delay=10,
        # Maximum allowed fan-in per target-layer neuron
        s_max=64,
        # Frequency of rewiring in Hz
        f_rew=10 ** 4,
        # timing and weight as required for neuromodulation
        timing_dependence=sim.SpikePairRule(
        tau_plus=2, tau_minus=1,
        A_plus=1, A_minus=1),
        weight_dependence=sim.AdditiveWeightDependence(w_min=0, w_max=5.0))
```

The remaining network is set up similarly to in the previous example, and the resulting spike raster on the population is shown in Figure 2.

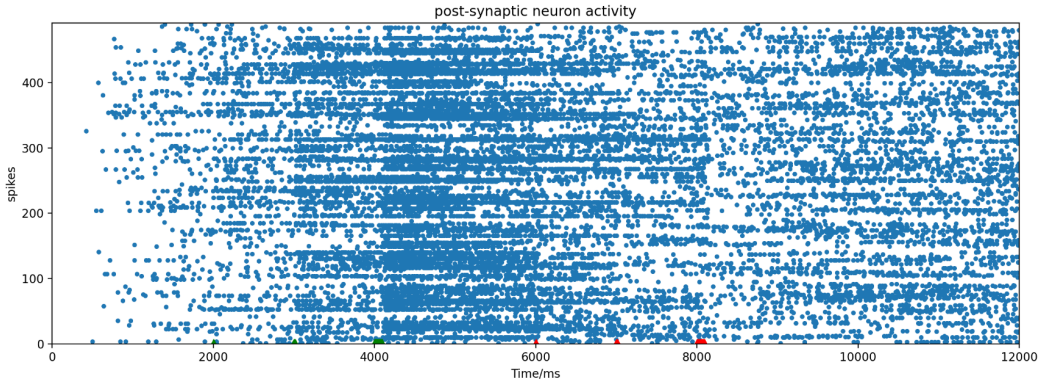

Figure 2: post-synaptic neuron activity across multiple populations affected by rewards (green) followed by punishments (red), with built-in rewiring (formation and elimination).

Note also that in this particular example, the binary with neuron update plus synapse update with both neuromodulation and structural plasticity is currently larger than the available instruction memory on a single core. We are able to still run an example like this because of other work within the code which allows the splitting of code onto “neuron cores” and “synapse cores”. This is achieved in this example by simply adding a (non-PyNN) splitter object to the post-population’s additional_parameters dictionary like so:

```
for i in range(n_pops):
        stimulation.append(sim.Population(n_neurons, sim.SpikeSourcePoisson,
```

```
                {'rate': stim_rate, 'duration': duration}, label="pre"))

    post_splitters.append(SplitterAbstractPopulationVertexNeuronsSynapses(1))
    post_pops.append(sim.Population(
           n_neurons, sim.IF_curr_exp, cell_params, label='post',
           additional_parameters={"splitter": post_splitters[i]}))
```

with the remainder of the PyNN script remaining the same as it would be without the use of the splitter. The argument of the splitter object details how many synapse cores there are per neuron core; in this instance we only need to choose 1 because we are only interested in being able to compile the code, but it is possible to choose a number of synapse cores per neuron core in this way that should only be limited by the cores having to be on the same chip as this code works through their shared (chip's) memory (SDRAM).

3. More Details

If you are interested in editing the code further, for example to add more implementation rules and/or synapse dynamics, then the place to start is to look at the current (Izhikevich-based) implementation in

`sPyNNaker/neural_modelling/src/neuron/plasticity/stdp/synapse_dynamics_izkhikevich_neuromodulation.c`

and also have a read of the paper describing the (initial) implementation of this [2].

References

[1] **Solving the distal reward problem through linkage of STDP and dopamine signaling** Eugene M Izhikevich, *Cerebral Cortex*, Volume 17, Issue 10, October 2007, DOI: 10.1093/cercor/bhl152

[2] **Neuromodulated Synaptic Plasticity on the SpiNNaker Neuromorphic System** Mantas Mikaitis, Garibaldi Pineda García, James C. Knight and Steve B. Furber, Front. Neurosci., 27 February 2018, DOI: https://doi.org/10.3389/fnins.2018.00105

[3] **sPyNNaker: A Software Package for Running PyNN Simulations on SpiNNaker** Oliver Rhodes, Petruţ A. Bogdan, Christian Brenninkmeijer, Simon Davidson, Donal Fellows, Andrew Gait, David R. Lester, Mantas Mikaitis, Luis A. Plana, Andrew G. D. Rowley, Alan B. Stokes and Steve B. Furber, Front. Neurosci., 20 November 2018, DOI: https://doi.org/10.3389/fnins.2018.00816

[4] **Structural Plasticity on the SpiNNaker Many-Core Neuromorphic System** Petruț A. Bogdan, Andrew G. D. Rowley, Oliver Rhodes and Steve B. Furber, Front. Neurosci., 02 July 2018 DOI: https://doi.org/10.3389/fnins.2018.00434