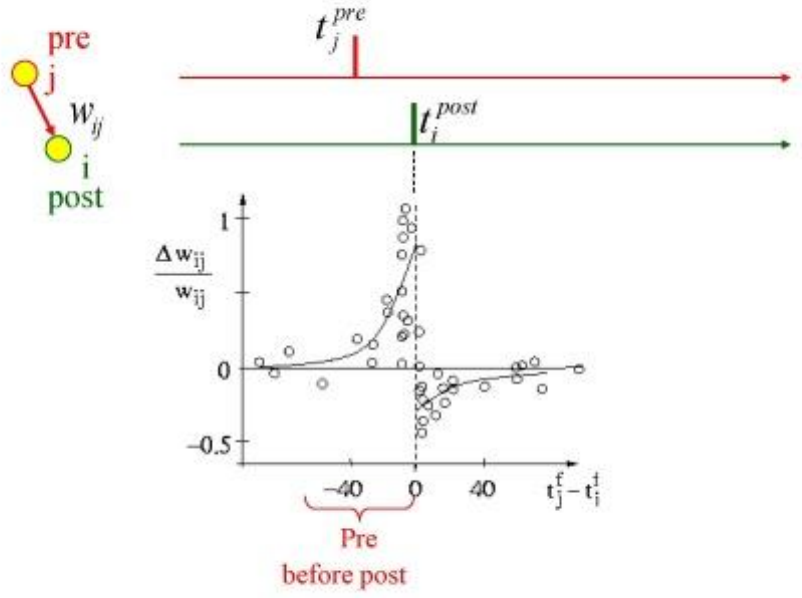
Spike-Time-Dependent Plasticity on SpiNNaker

STDP is a form of learning that depends upon the timing between the spikes of two neurons connected by a synapse. It is believed to be the basis of learning and information storage in the human brain.

In the case where a presynaptic spike is followed closely by a postsynaptic spike, then it is presumed that the presynaptic neuron caused the spike in the postsynaptic neuron, and so the weight of the synapse between the neurons is increased. This is known as potentiation.

If a postsynaptic spike is emitted shortly before a presynaptic spike is emitted, then the presynaptic spike cannot have caused the postsynaptic spike, and so the weight of the synapse between the neurons is reduced. This is known as depression.

The size of the weight change depends on the relative timing of the presynaptic and postsynaptic spikes; in general, the change in weight drops off exponentially as the time between the spikes gets larger, as shown in the following figure [Sjöström and Gerstner (2010), Scholarpedia]. However, 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 presynaptic and postsynaptic spikes to trigger synaptic potentiation or depression.



STDP in PyNN

The steps for creating a network using STDP are much the same as previously described, with the main difference being that some of the projections use a `STDPMechanism` to describe the plasticity. Here is an example of the creation of a projection with STDP:

```
timing_rule = sim.SpikePairRule(tau_plus=20.0, tau_minus=20.0,
                                 A_plus=0.5, A_minus=0.5)
weight_rule = sim.AdditiveWeightDependence(w_max=5.0, w_min=0.0)

stdp_model = sim.STDPMechanism(timing_dependence=timing_rule,
                               weight_dependence=weight_rule,
                               weight=0.0, delay=5.0)

stdp_projection = sim.Projection(pre_pop, post_pop, sim.OneToOneConnector(),
                                 synapse_type=stdp_model)
```

In this example, firstly the timing rule is created. In this case, it is a *SpikePairRule*, which means that the

relative timing of the spikes that will be used to update the weights will be based on pairs of presynaptic and postsynaptic spikes. This rule has four parameters. The parameters *tau_plus* and *tau_minus* describe the respective exponential decay of the size of the weight update with the time between presynaptic and postsynaptic spikes. Note that the decay can be different for potentiation (defined by *tau_plus*) and depression (defined by *tau_minus*). The parameters *A_plus* and *A_minus* which define the maximum weight to respectively add during potentiation or subtract during depression.

The next thing defined is the weight update rule. In this case it is a *AdditiveWeightDependence*, which means that the weight will be updated by simply adding to the current weight. This rule requires the parameters *w_max* and *w_min*, which define the maximum and minimum weight of the synapse respectively, Note that the actual amount added or subtracted will depend additionally on the timing of the spikes, as determined by the timing rule.

In addition, there is also a *MultiplicativeWeightDependence* supported, which means that the weight change depends on the difference between the current weight and w_max for potentiation, and w_min for depression. The value of *A_plus* and *A_minus* are then respectively multiplied by this difference to give the maximum weight change; again the actual value depends on the timing rule and the time between the spikes.

The timing and weight rules are combined with *weight* and *delay* into a single *STDPMechanism* object which describes the overall desired mechanism. Note that the projection still requires the specification of a *connector.* This connector is still used to describe the overall connectivity between the neurons of the pre- and post-populations. It is preferable that the initial weights fall between *w_min* and *w_max*; it is not an error if they do not, but when the first update is performed, the weight will be changed to fall within this range.

Note that on SpiNNaker, although multiple projections to the same target population can be specified with STDP, the restrictions on the current software are that all those projections must use the same rules with the exact same parameters. This is due to the restrictions of the local memory available on each core, reducing the amount of data that can be held for the parameters.

Note: In the implementation of STDP on SpiNNaker, the plasticity mechanism is only activated when the second presynaptic spike is received at the postsynaptic neuron. Thus at least two presynaptic spikes are required for the mechanism to be activated.

Getting Synaptic Data

The weights and delays assigned to a projection can be retrieved using the Projection's *get* method, specifying the data items to get, including 'weight', 'delay' and the parameters of the STDP Mechanism, and the format they are retrieved using. The data formats supported are 'list' format, where the return value consists of a list of tuples of the selected values; and 'array' where each value is returned in a two-dimensional matrix indexed by the source neurons in the pre-population, and the target neurons in the post-populations. In the 'list' result, each tuple additionally contains the source and target neuron ids as the 0th and 1st values in the tuple. In the 'array' result, missing connections are represented as 'NaN' (not a number) and positions where there are multiple connections have their values summed.

Note that on SpiNNaker, it is possible to retrieve the projection data before calling the PyNN run function, but that this data cannot be examined until after run has been called. This is because the individual connectivity data is not generated until the run function is called.

Task 1: Explore Existing STDP Implementations - Repeated from Lab 02

Task 1.1: STDP Network [Easy]

This task will create a simple network involving STDP learning rules.

Write a network with a 1.0ms time step consisting of two single-neuron populations connected with an STDP synapse using a spike pair rule and additive weight dependency, and initial weights of 0. Stimulate each of the neurons with a spike source array with times of your choice, with the times for stimulating the first neuron being slightly before the times stimulating the second neuron (e.g. 2ms or more), ensuring the times are far enough apart not to cause depression (compare the spacing in time with the tau_plus and tau_minus settings); note that a weight of 5.0 should be enough to force an IF_curr_exp neuron to fire with the default parameters. Add a few extra times at the end of the run for stimulating the first neuron. Run the network for a number of milliseconds and extract the spike times of the neurons and the weights.

You should be able to see that the weights have changed from the starting values, and that by the end of the simulation, the second neuron should spike shortly after the first neuron.

Task 1.2: STDP Parameters [Easy]

Alter the parameters of the STDP connection, and the relative timing of the spikes. Try starting with a large initial weight and see if you can get the weight to reduce using the relative timing of the spikes.

Task 1.3: STDP Curve [Hard]

This task will attempt to plot an STDP curve, showing how the weight change varies with timing between spikes.

1. Set up the simulation to use a 1ms time step.
2. Create a population of 100 presynaptic neurons.
3. Create a spike source array population of 100 sources connected to the presynaptic population. Set the spikes in the arrays so that each spikes twice 200ms apart, and that the first spike for each is 1ms after the first spike of the last e.g. [[0, 200], [1, 201], …] (hint: you can do this with a list comprehension).
4. Create a population of 100 postsynaptic neurons.
5. Create a spike source array connected to the postsynaptic neurons all spiking at 50ms.
6. Connect the presynaptic population to the postsynaptic population with an STDP projection with an initial weight of 0.5 and a maximum of 1 and minimum of 0.
7. Record the presynaptic and postsynaptic populations.
8. Run the simulation for long enough for all spikes to occur, and get the weights from the STDP projection.
9. Draw a graph of the weight changes from the initial weight value against the difference in presynaptic and postsynaptic neurons (hint: the presynaptic neurons should spike twice but the postsynaptic should only spike once; you are looking for the first spike from each presynaptic neuron).

Task 2: Create New STDP Rule [Medium]

This task will create a custom STDP mechanism based on existing spike pair timing and additive weight dependence rules. In the following task, these rules can then be edited to customise learning.

Task 2.1 Create Custom Timing Dependence Rule

1. First address the python code
    a. Find timing_dependence_spike_pair.py and copy to a new name in the same directory e.g. timing_dependence_spike_pair_custom.py
    b. Update the class name to reflect the new name: e.g. TimingDependenceSpikePairCustom
    c. The base spike timing dependence rule in sPyNNaker was developed to fulfil the requirements of both PyNN 0.7 and PyNN 0.8 (specifically in how the a_plus and a_minus parameters are passed in - see lecture slides for more details). As this is now a custom timing rule, it's no longer subject to the requirements of PyNN, and hence the class can be edited to accept the a_plus and a_minus parameters directly. This simplifies execution, and serves as an example of how to pass in parameters to the timing rule.

    Edit the python class to take A_plus and A_minus on the __init__ function, and set these as instance parameters (note capitalisation):

```
class TimingDependenceSpikePairCustom(AbstractTimingDependence):

    def __init__(self, tau_plus=20.0, tau_minus=20.0, A_plus=0.01, A_minus=0.01):
        AbstractTimingDependence.__init__(self)
        self._tau_plus = tau_plus
        self._tau_minus = tau_minus

        self.A_plus = A_plus
        self.A_minus = A_minus

        self._synapse_structure = SynapseStructureWeightOnly()
```

    d. Edit the `vertex_executable_suffix(self)` function to return a suffix associated with the new timing rule - e.g. "pair_custom"
    e. Edit the `is_same_as(self, timing_dependence)` function to compare to the new class name:

```
if not isinstance(timing_dependence, TimingDependenceSpikePairCustom):
```

    f. To make the new rule available through the PyNN interface, edit the adjacent __init__.py file to import the new timing rule - e.g. add:

```
from .timing_dependence_spike_pair_custom import
TimingDependenceSpikePairCustom

__all__ = ["AbstractTimingDependence", "TimingDependenceSpikePair",
           "TimingDependencePfisterSpikeTriplet", "TimingDependenceRecurrent",
           "TimingDependenceSpikeNearestPair", "TimingDependenceVogels2011",
           "TimingDependenceSpikePairCustom"]
```

g. Now add similar import statements to the sPyNNaker8 repository. Find spynnaker8/__init__.py and add:

```
from spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence \
    .timing_dependence_spike_pair_custom \
    import TimingDependenceSpikePairCustom as SpikePairRuleCustom

__all__ = [...,
      # plastic stuff
      'STDPMechanism', 'AdditiveWeightDependence', 'SpikePairRule',
      'MultiplicativeWeightDependence','SpikePairRuleCustom', ...]
```

2. Now update the C code
    a. Find timing_pair_impl.c and timing_pair_impl.h and copy to new locations in the same directory: e.g. timing_pair_custom_impl.c and timing_pair_custom_impl.h
    b. In the C builds directory, create a new directory for your custom build: here the IF_curr_exp neuron model will be used, so the name should start with this. When searching for a binary, the toolchain will add the suffix "_stdp_mad_" to the neuron model, together with the suffix from your timing and weight dependence classes. Therefore, for the current example using the existing weight dependence rule, the build directory should be named: **IF_curr_exp_stdp_mad_pair_custom_additive**
    c. Copy the Makefile from the adjacent IF_curr_exp_stdp_mad_pair_additive directory, and edit the paths of the TIMING_DEPENDENCE and TIMING_DEPENDENCE_H variables to point to the custom .c and .h files created in step a.
    d. Now edit Makefile in the 'Neuron' directory (two levels up from the build directory), adding your new build directory to the end of the list defined by the variable MODELS
    e. You should now be able to navigate back to the build directory and type 'make' to build the new binary - if the build was successful the last few logging lines will report the name of the newly built aplx file (IF_curr_exp_stdp_mad_pair_custom_additive.aplx)

3. Run test example
    a. In one of the PyNN scripts used in section 1, change the STDPMechanism to use the new timing rule: timing_dependence=p.**SpikePairRuleCustom**(...
    b. Re-run the script to test execution

Task 2.2 Create Custom Weight Dependence Rule

1. First address the Python code
    a. Find weight_dependence_additive.py and copy to a new name in the same directory e.g. weight_dependence_additive_custom.py
    b. Update the class name to reflect the new name: e.g. WeightDependenceAdditiveCustom
    c. Edit the `vertex_executable_suffix(self)` function to return a suffix associated with the new weight dependence rule: e.g. `"additive_custom"`
    d. Edit the `is_same_as(self, timing_dependence)` function to compare to the new class name:

```
if not isinstance(weight_dependence, WeightDependenceAdditiveCustom):
```

e. Edit the adjacent __init__ file to import the new timing rule; e.g.

```
from .weight_dependence_additive_custom import WeightDependenceAdditiveCustom

__all__ = ["AbstractHasAPlusAMinus", "AbstractWeightDependence",
           "WeightDependenceAdditive", "WeightDependenceMultiplicative",
           "WeightDependenceAdditiveTriplet",
"WeightDependenceAdditiveCustom"]
```

f. Now add similar import statements to the sPyNNaker8 repository. Find spynnaker8/__init__.py and add:

```
from spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence \
    .weight_dependence_additive_custom\
    import WeightDependenceAdditiveCustom as
AdditiveWeightDependenceCustom

__all__ = [...,
      # plastic stuff
      'STDPMechanism', 'AdditiveWeightDependence',
      'AdditiveWeightDependenceCustom', 'MultiplicativeWeightDependence',
      'SpikePairRule', 'SpikePairRuleCustom', ...]
```

2. Now update the C Code
   a. Find weight_additive_one_term_impl.h and weight_additive_one_term_impl.c and copy to new names in the same directory: e.g. weight_additive_one_term_custom_impl.h and weight_additive_one_term_additive_custom_impl.c
   b. In the C builds directory, create a new directory to house the build for the new application including both the custom timing and weight dependence rules. Hint, take the name of the build directory from Task 2.1, and adjust the weight suffix to the custom rule: e.g. IF_curr_exp_stdp_mad_pair_custom_weight_custom
   c. Copy the Makefile from Task 2.1.2-c, and update the paths for WEIGHT_DEPENDENCE and WEIGHT_DEPENDENCE_H, e.g. to point to weight_additive_one_term_custom_impl.h and weight_additive_one_term_custom_impl.c respectively
   d. As in Task 2.1.2-d, edit the Makefile in the neuron directory (two levels up from current Makefile) to include the new neuron model including both custom timing and weight dependence rules: IF_curr_exp_stdp_mad_pair_custom_weight_custom
   e. Run ‘make’ to build the application, and check the aplx name matches that defined by the Python code

3. Run test example
   a. In the PyNN scripts used for task 2.1.3, change the STDPMechanism to also use the new weight rule: timing_dependence=p.**WeightDependenceAdditiveCustom**(...
   b. Re-run the script and check execution

Task 3: Customise Timing and Weight Dependence Rules to Modify Learning [As Hard As You Make IT]

Implement your own ideas for timing and weight dependence rules by editing the new custom configurations, or alternatively follow the steps below to add an extra variable to the timing rule.

1. Adding a new variable to the timing dependence rule
    a. Edit the Python code to ensure the new variable is accessible through PyNN, and that the toolchain will write the new parameter ready for loading (hint: ensure the correct size is specified, too)
    b. Edit the C source to read the new parameter at initialisation
    c. Use the parameter in one of the spike handling functions in the timing rule header
    d. Re-compile the binary
    e. Write test PyNN script to check operation