Using and extending plasticity rules in Nengo

Trevor Bekolay

Centre for Theoretical Neuroscience technical report. Sept 10, 2010

Abstract

Learning in the form of synaptic plasticity is an essential part of any neural simulation software claiming to be biologically plausible. While plasticity has been a part of Nengo from the beginning, few simulations have been created to make full use of the plasticity mechanisms built into Nengo, and as a result, they have been under-maintained. Since SVN revision 985, the way plasticity is implemented in Nengo has changed significantly. This report is intended to explain how plasticity rules are implemented since that revision, and provide examples of how to use and extend the plasticity rules currently implemented.

1 Introduction

Learning in neural systems is a rich area of on-going research. The majority of work deals with synaptic plasticity, the notion that the strength of the connection between a presynaptic neuron and postsynaptic neuron changes in certain situations [8,9] . A recent review paper suggests that synaptic plasticity is necessary for learning and memory, but is not necessarily sufficient [1].

The situations under which synaptic connection weights change are made explicit with *learning rules*, which define how the strength of a particular synapse changes as a function of a number of factors. Many decades of experimental work and early modeling efforts have produced a wide variety of learning rules, such as spike-timing dependent plasticity [3, 5] and the BCM rule [4].

The Neural Engineering Framework (NEF) [6], and the Nengo modelling software based on the NEF, is uniquely poised as a robust environment in which to implement and test learning rules. This is in a sense contrary to the purpose of the NEF, which defines an analytical method of finding connection weight matrices that will transform signals. Learning is not only a completely different way of finding an appropriate connection weight matrix, it is also less explanatory, as the transformation performed by a learned connection weight matrix can be difficult to characterize.

However, doing learning in Nengo is still more explanatory than traditional simulations of learning in biologically inspired neural networks. The following are some reasons that we might want to use Nengo for doing simulations of plastic systems.

- Nengo is able to simulate at different levels of granularity. We can test learning rules defined for rate-based models and learning rules that depend on precise spike times.
- Principle 1 of the NEF, encoding, allows us a method to examine the types of functions that different learning rules can learn.
- Principle 2 of the NEF, transformation, provides an analytical method of producing a connection weight matrix to perform a certain transformation. Comparing that "optimal" matrix to a learned matrix should elucidate features of the learning rule, the NEF, or both.

- Learning can help fine-tune existing models.

Some of these reasons were noted in [6], the most complete description of the NEF. In that book, a typical Hebbian learning rule is given in "delta rule" form, the form most commonly used in learning literature. The rule given in the book is

$$\Delta\omega_{ij} = -\kappa\left(b_j(x) > 0\right)a_i(x)(b_j(x) - \bar{b}_j(x)\right) .$$

We will see how to implement this learning rule in Nengo in sections 2 and 3. The delta-rule equation is known as the learning function, and is examined in detail in section 2. Section 3 describes plasticity rules, which define when a learning function should be applied. Section 4 goes over other pieces of code that are relevant to plasticity in Nengo. Finally, Appendix A details the changes that were made in SVN revision 985; this is included for completeness, and is unlikely to be useful for anyone who has not used plasticity rules prior to SVN revision 985.

2 Plasticity rules in Nengo

When examining literature on plasticity in neural simulations, many similar terms are used with little attention given to their precise definitions. In particular, the following terms are used synonymously in most cases.

- Plasticity rule, plasticity function
- Learning rule, learning function
- Delta rule, delta function

All of these terms typically refer to an expression like

$$\Delta\omega_{ij} = -\kappa\left(b_j(x) > 0\right)a_i(x)(b_j(x) - \bar{b}_j(x)\right) .$$

The left side of the expression, $\Delta\omega_{ij}$, is common to all learning rules.

- ω is the connnection weight matrix; i.e. a matrix of real numbers that represent the strength of the connection between presynaptic neurons and postsynaptic neurons.
- ω_{ij} is the real number found in the ith row and jth column in the ω matrix.
- $\Delta\omega_{ij}$ is the change in value of ω_{ij}. $\Delta\omega_{ij}$ may be computed each timestep, each spike, or when some other condition holds true.

The right side of the equation is different for each learning rule; most papers will explicitly state what each variable on the right side of the equation represents. Some variables, however, show up in most – though not all – cases.

- κ is the learning rate. This is a scalar value that every $\Delta\omega_{ij}$ will be scaled by to ensure that the rule learns at an appropriate speed.
- $a_i(x)$, $b_j(x)$ and other similar variables typically refer to the activity of a certain neuron. These are typically found in rules used in rate-based models.

- t^{post}, t^{pre}, and associated τ values represent times of post and pre spikes, and the length of the active window in rules used in spike-timing dependent models.

These conventions are maintained whenever possible in Nengo. Where convention is not followed, it is typically for reasons of clarity; since Nengo is software, it is afforded more space for variable names. For example, in the above equation, i and j typically refer to the index of the presynaptic and postsynaptic neuron, respectively. However, the opposite convention could be followed just as easily, so in Nengo these indices are referred to as `preIndex` and `postIndex`.

Nengo also differentiates between **learning functions** and **plasticity rules**. An equation of the form $\Delta\omega_{ij} = ...$ does not give a complete description of learning, because it does not define **when** the change in synaptic strength is applied; that information is typically given in the text of a learning paper.

In Nengo, the $\Delta\omega_{ij}$ rule is called the *learning function*. When the learning function is applied is defined by the *plasticity rule*.

In addition to being necessary, this separation is useful because there are many different "learning functions," but only a few "plasticity rules." This makes implementing new learning rules in Nengo quite easy, as one only needs to create a new learning function.

3 Learning functions

In Nengo, a learning function conforms to the `Function` template (defined in `ca.nengo.math.Function`). It is a subclass of `AbstractFunction` (`ca.nengo.math.impl.AbstractFunction`). The only thing that a programmer needs to know about these classes is that, by using this class hierarchy, it makes the dimensionality of a learning function immutable.

In Nengo, there are Java classes for two types of learning functions: those used in rate-based simulations, and those used in spiking simulations.

3.1 Real learning functions

For rate-based simulations, the learning function used will be a subclass of `AbstractRealLearningFunction` (`ca.nengo.plasticity.impl.AbstractRealLearningFunction`). The class is abstract, which means that it cannot be instantiated; it must be subclassed.

The only abstract method – that is, the only method that must be provided by a subclass of `AbstractRealLearningFunction` – is `protected abstract float deltaOmega(·)`. Note the name of the function – this makes it clear that "learning functions" implement the $\Delta\omega_{ij}$ portion of a learning rule. This function accepts eight arguments.

1. `float input` The activity coming into the synapse from its inputs.
2. `float time` The current simulation time.
3. `float currentWeight` The current connection weight between the pre and post neurons.
4. `float modInput` The modulatory input, for this particular dimension (the dimension in question is another input to this function).
5. `float originState` The state of the origin from the postsynaptic population, for this particular dimension.

6. `int postIndex` The neuron index in the post-synaptic population.
7. `int preIndex` The neuron index in the pre-synaptic population.
8. `int dim` The dimension of the modulatory input and the origin state.

These eight arguments represent all of the information that will change as the simulation proceeds. Sometimes other arguments are required by a learning function; however, those arguments typically do not change over the course of a simulation.

At the time this report was written, one non-abstract real learning function was implemented in Nengo, and it serves as a good example for other real learning functions that require other non-changing arguments.

The non-abstract real learning function is `ErrorLearningFunction` (`ca.nengo.plasticity.impl.ErrorLearningFunction`). This class implements the following learning rule, described in [7].

$$\Delta\omega_{ij} = \kappa\alpha_j\tilde{\phi}_j\mathbf{e}a_i$$

where $\mathbf{e}$ is a global error signal, and α and $\tilde{\phi}$ are the gain and encoder of the output population, following NEF conventions. α and $\tilde{\phi}$ are immutable properties associated with neurons in a population. Because they don't change, they are passed to the `ErrorLearningFunction` in its constructor.

3.1.1 Using real learning functions

The only learning rule that can be used without creating a new class is the `ErrorLearningFunction`. Below is an example of creating an `ErrorLearningFunction` in a Python script. Note that this script will not run anything useful, it is only intended to show the correct syntax for instantiating the `ErrorLearningFunction`.

```
ef = NEFEnsembleFactoryImpl()
pre = ef.make('pre', 100, 1) # 100 neurons, 1 dimension
post = ef.make('post', 200, 1) # 200 neurons, 1 dimension
learnFcn = ErrorLearningFunction([n.scale for n in post.nodes], post.encoders)
```

3.1.2 Extending real learning functions

For any other real learning function, the `AbstractRealLearningFunction` must be subclassed. (`ErrorLearningFunction` or any other implemented real learning function can also subclassed if appropriate.) In this section, we will use the following simple learning function as an example.

$$\Delta\omega_{ij} = \kappa a_i(x)b_j(x)$$

Ideally, the learning function would be implemented in Java in the `ca.nengo.plasticity.impl` package so that the learning rule can be used easily in the future. Below is the appropriate Java code for implementing the above function in Java. For a full example, including import statements and appropriate comments, see `ErrorLearningFunction`.

```java
public class SimpleRealLearningFunction extends AbstractRealLearningFunction {

    protected float deltaOmega(float input, float time, float currentWeight,
            float modInput, float originState, int postIndex, int preIndex, int dim) {
        return myLearningRate * input * originState;
    }
}
```

However, for very simple learning rules, or for quick prototyping, the rule can be subclassed in a Python script as well.

```python
class SimpleRealLearningFunction(AbstractRealLearningFunction)
    def deltaOmega(self, input, time, current_weight, mod_input, origin_state,
            post_index, pre_index, dim):
        return self.myLearningRate * input * origin_state
```

If any additional variables are needed, they should be stored as member variables in your subclass. The member variables should be instantiated in the constructor of your subclass, unless there is a reason that they cannot be instantiated in the constructor. For an example of how to do this in Java, see `ErrorLearningFunction`. An equivalent Python implementation of that Java class is below.

```python
class ErrorLearningFunction(AbstractRealLearningFunction)
    def __init__(self,gain,encoders):
        AbstractRealLearningFunction.__init__()
        self.gain = gain
        self.encoders = encoders

    def deltaOmega(self, input, time, current_weight, mod_input, origin_state,
            post_index, pre_index, dim):
        return self.myLearningRate * (input * mod_input * self.encoders[post_index][dim] *
            self.gain[post_index]  - (origin_state*origin_state*current_weight))
```

3.2 Spike learning functions

For spike-timing dependent simulations, the learning function used will be a subclass of `AbstractSpikeLearningFunction` (`ca.nengo.plasticity.impl.AbstractSpikeLearningFunction`). The class is abstract, which means that it cannot be instantiated; it must be subclassed.

The only abstract method – that is, the only method that must be provided by a subclass of `AbstractSpikeLearningFunction` – is `protected abstract float deltaOmega(·)`. This function accepts seven arguments.

1. `float timeSinceDifferent` The amount of time passed since the last spike of the different type – that is, if this is an onInSpike function, it would be the amount of time since the last out spike.
2. `float timeSinceSame` The amount of time passed since the last spike of the same type – that is, if this is an onInSpike function, it would be the amount of time since the last in spike.
3. `float currentWeight` The current connection weight between the pre and post neurons.
4. `float modInput` The modulatory input, for this particular dimension (see dim).

5. `int postIndex` The neuron index in the post-synaptic population.
6. `int preIndex` The neuron index in the pre-synaptic population.
7. `int dim` The dimension of the modulatory input.

Examining `AbstractSpikeLearningFunction` reveals two important differences between spike-timing dependent learning functions and real learning functions. First, the arguments to the `deltaOmega` function are significantly different. Second, spike-timing dependent learning rules have *two* learning functions: one that is used when the presynaptic neuron spikes, and one that is used when the postsynaptic neuron spikes. This is made explicit in the associated plasticity rule (see section 4).

At the time this report was written, two non-abstract spike learning functions are implemented in Nengo. The non-abstract real learning functions are `InSpikeErrorFunction` (`ca.nengo.plasticity.impl.InSpikeErrorFunction`) and `OutSpikeErrorFunction` (`ca.nengo.plasticity.impl.OutSpikeErrorFunction`). These classes implement the following learning rules, described in [2].

For a presynaptic spike (in spike):

$$\Delta\omega_{ij}(t^{pre}) = \kappa\alpha_j\tilde{\phi}_j e\left(-e^{\frac{-t(^{pre}-t^{post_1})}{\tau_-}}\left[A_2^- + A_3^- e^{\frac{-(t^{pre}-t^{pre_2})}{\tau_x}}\right]\right).$$

For a postsynaptic spike (out spike):

$$\Delta\omega_{ij}(t^{post}) = \kappa\alpha_j\tilde{\phi}_j e\left(e^{\frac{-(t^{post}-t^{pre_1})}{\tau_+}}\left[A_2^+ + A_3^+ e^{\frac{-(t^{post}-t^{post_2})}{\tau_y}}\right]\right).$$

For details on what the variables mean, please see [2].

3.2.1 Using spike learning functions

The only learning rules that can be used without creating a new class are the `InSpikeErrorFunction` and `OutSpikeErrorFunction`. Below is an example of creating these functions in a Python script. Note that this script will not run anything useful, it is only intended to show the correct syntax for instantiating the spike learning functions.

```
ef = NEFEnsembleFactoryImpl()
pre = ef.make('pre', 100, 1) # 100 neurons, 1 dimension
post = ef.make('post', 200, 1) # 200 neurons, 1 dimension
inFcn = InSpikeErrorFunction([n.scale for n in post.nodes],post.encoders)
outFcn = OutSpikeErrorFunction([n.scale for n in post.nodes],post.encoders)
```

3.2.2 Extending spike learning functions

For any other spike learning function, the `AbstractSpikeLearningFunction` must be subclassed. (`InSpikeErrorFunction` or any other implemented spike learning function can also subclassed if appropriate.) In this section, we will use the following simple spike-timing dependent learning function as an example.

$$\Delta\omega_{ij}(t^{pre}) = \kappa(t^{pre} - t^{post})$$

Ideally, the learning function would be implemented in Java in the `ca.nengo.plasticity.impl` package so that the learning rule can be used easily in the future. Below is the appropriate Java code for implementing

the above function in Java. For a full example, including import statements and appropriate comments, see `InSpikeErrorFunction`.

```
public class SimpleSpikeLearningFunction extends AbstractSpikeLearningFunction {

    protected float deltaOmega(float timeSinceDifferent, float timeSinceSame,
            float currentWeight, float modInput, int postIndex, int preIndex, int dim) {
        return myLearningRate * timeSinceDifferent;
    }
}
```

However, for very simple learning rules, or for quick prototyping, the rule can be subclassed in a Python script as well.

```
class SimpleSpikeLearningFunction(AbstractSpikeLearningFunction)
    def deltaOmega(self, time_since_different, time_since_same, current_weight,
            mod_input, post_index, pre_index, dim):
        return self.myLearningRate * time_since_different
```

If any additional variables or constants are needed, they should be stored as members of subclass. The member variables should be instantiated in the constructor of your subclass, unless there is a reason that they cannot be instantiated in the constructor. For an example of how to do this in Java, see `InSpikeErrorFunction`. An equivalent Python implementation of that Java class is below.

```
class InSpikeErrorFunction(AbstractSpikeLearningFunction)
    def __init__(self, gain, encoders, a2minus=6.6e-3, a3minus=3.1e-3,
            tau_minus=0.0337, tau_x=0.714):
        AbstractSpikeLearningFunction.__init__()
        self.gain = gain
        self.encoders = encoders
        self.a2minus = a2minus
        self.a3minus = a3minus
        self.tau_minus = tau_minus
        self.tau_x = tau_x

    def deltaOmega(self, time_since_different, time_since_same, current_weight,
            mod_input, post_index, pre_index, dim):

        o1 = 0.0
        if (time_since_different  self.tau_minus):
            o1 = math.exp(-time_since_different/self.tau_minus)

        r2 = 0.0
        if (time_since_same  self.tau_x):
            r2 = math.exp(-time_since_same/self.tau_x)

        result = o1 * (self.a2minus + r2 * self.a3minus)

        return myLearningRate * result * mod_input *
            self.encoders[post_index][dim] * self.gain[post_index];
```

4 Plasticity rules

In Nengo, a "plasticity rule" is a class that implements the `PlasticityRule` interface, defined in `ca.nengo.plasticity.PlasticityRule`. The `PlasticityRule` interface requires a plasticity rule to implement the following three functions.

`public void setModTerminationState(String name, InstantaneousOutput state, float time)`
: This function will be called before the learning function; it is used to set the modulatory input that will be used in the learning function (`modInput` in the real and spike learning functions).

`public void setOriginState(String name, InstantaneousOutput state, float time)`
: This function will be called before the learning function; it is used to set the output of the population that is undergoing learning (`originState` in the real learning functions). In the spike learning function, this function updates the spiking status of the output neurons and updates the spike history.

`public float[][] getDerivative(float[][] transform, InstantaneousOutput input, float time)`
: This function is the one that calls the learning function or functions (the $\Delta\omega_{ij}$ portion of a learning rule). It is provided the current ω weight matrix, the input to the termination that is having its weights learned, and the current time.

While a learning function defines the $\Delta\omega_{ij}$ portion of a learning rule, the plasticity rule defines when that learning function will be applied. We have identified two situations in which the learning function is applied at different times. These situations have been implemented as `RealPlasticityRule` and `SpikePlasticityRule`. If you want to make your simulation adaptable to both situations, the `CompositePlasticityRule` allows you to use both a real and spike plasticity rule. There are, however, speed tradeoffs for using this approach.

4.1 Using RealPlasticityRule

`RealPlasticityRule` (`ca.nengo.plasticity.impl.RealPlasticityRule`) applies the learning function at every timestep of the simulation. It is intended to be used with rate-based simulations, or in other words, terminations that transfer data using the `RealOutput` class. Because of this, the `RealPlasticityRule` requires that the learning function it uses is a subclass of `AbstractRealLearningFunction`.

To use `RealPlasticityRule`, the only thing that must be understood is its constructor.

`public RealPlasticityRule(AbstractRealLearningFunction function, String originName, String modTermName)`
: The `RealPlasticityRule` constructor requires an `AbstractRealLearningFunction`, which is the learning function that will be applied one each timestep, a string representing the name of the origin that should be used to gather activity from the output population (usually 'AXON'), and a string representing the termination that will be provided modulatory input. Note that the modulatory termination must be a `DecodedTermination`, and that if no modulatory input is being used, can be `null`.

For further details on exactly how `RealPlasticityRule` works, examine the `getDerivative` function. Note that, if modulatory input is present, there are three nested loops: one that loops over each presynaptic neuron, one that loops over each postsynaptic neuron, and one that loops over each dimension in the modulatory input. So, if we were to learn weights from a population with 80 neurons to a population with 120 neurons, and there is a 2-dimensional modulatory input, we will be calling the learning function $120 \times 80 \times 2 = 19200$ times every timestep. If no modulatory input is being used by the rule, then there are only two nested loops, one for each presynaptic neuron and one for each postsynaptic neuron.

A minimal Python example for using the `RealPlasticityRule` with a learning function already implemented in Nengo follows. This code snippet builds on the code presented in section 3.1.1. Like that code snippet,

this one will not run anything useful, it is only intended to show the correct syntax for instantiating and applying the `RealPlasticityRule`.

```
ef = NEFEnsembleFactoryImpl()
pre = ef.make('pre', 100, 1) # 100 neurons, 1 dimension
post = ef.make('post', 200, 1) # 200 neurons, 1 dimension
# Add a termination for modulatory input
post.addDecodedTermination('mod', [[1]], 0.005, True)
# Add a non-decoded termination that will be learned
post.addTermination('learn', weights, 0.005, False)
error = ef.make('error', 150, 1) # 150 neurons, 1 dimension
learnFcn = ErrorLearningFunction([n.scale for n in post.nodes], post.encoders)
rule = RealPlasticityRule(learnFcn, 'AXON', 'mod')
post.setPlasticityRule('learn', rule)
```

If the learning function you wish to use is not implemented in the Java source code, you can easily define it within the same Python script that uses your learning rule. The following example uses the `SimpleRealLearningFunction` described in section 3.1.2.

```
class SimpleRealLearningFunction(AbstractRealLearningFunction)
    def deltaOmega(self, input, time, current_weight, mod_input, origin_state,
            post_index, pre_index, dim):
        return self.myLearningRate * input * origin_state

...

ef = NEFEnsembleFactoryImpl()
pre = ef.make('pre', 100, 1) # 100 neurons, 1 dimension
post = ef.make('post', 200, 1) # 200 neurons, 1 dimension
# Add a non-decoded termination that will be learned
post.addTermination('learn', weights, 0.005, False)
simpleLearnFcn = SimpleRealLearningFunction()
rule = RealPlasticityRule(simpleLearnFcn, 'AXON', None) # Note no modulatory input
post.setPlasticityRule('learn', rule)
```

4.2 Using SpikePlasticityRule

`SpikePlasticityRule` (`ca.nengo.plasticity.impl.SpikePlasticityRule`) applies the learning function whenever a neuron in the presynaptic or postsynaptic population spikes. It is intended to be used with spiking simulations, or in other words, terminations that transfer data using the `SpikeOutput` class. Because of this, the `SpikePlasticityRule` requires that the learning functions it uses are a subclass of `AbstractSpikeLearningFunction`. A distinction is made between presynaptic and postsynaptic spikes, so this plasticity rule requires two different spike learning functions.

To use `SpikePlasticityRule`, the only thing that must be understood is its constructor. *Note: ASLF below stands for AbstractSpikeLearningFunction, shortened for space.*

`public SpikePlasticityRule(ASLF onInSpike, ASLF onOutSpike, String originName, String modTermName)`
: The `SpikePlasticityRule` constructor requires two `AbstractSpikeLearningFunctions`, one which will be applied at every presynaptic spike (or each “in spike”) and one which will be applied at every postsynaptic spike (or each “out spike”), a string representing the name of the origin that should be used

to gather activity from the output population (usually 'AXON'), and a string representing the termination that will be provided modulatory input. Note that the modulatory termination must be a `DecodedTermination`, and that if no modulatory input is being used, can be `null`.

For further details on exactly how `SpikePlasticityRule` works, examine the `getDerivative` function. Note that, if modulatory input is present, there are three nested loops: one that loops over each presynaptic neuron, one that loops over each postsynaptic neuron, and one that loops over each dimension in the modulatory input. If no modulatory input is being used by the rule, then there are only two nested loops, one for each presynaptic neuron and one for each postsynaptic neuron.

Unlike the `RealPlasticityRule`, the appropriate learning function will only be called if a neuron is spiking. So, if half of the neurons are spiking, both plasticity rules will call a learning function the same number of times. If no neurons are spiking, `SpikePlasticityRule` will not call any learning functions. If all of the neurons are spiking, `SpikePlasticityRule` will call twice as many learning functions as the `RealPlasticityRule`. In practice, there are usually less than half of the neurons spiking at any one time, so the `SpikePlasticityRule` may be more efficient. Of course, this is offset by the cost of doing a spiking simulation as opposed to a rate-based simulation.

A minimal Python example for using the `SpikePlasticityRule` with a learning function already implemented in Nengo follows. This code snippet builds on the code presented in section 3.2.1. Like that code snippet, this one will not run anything useful, it is only intended to show the correct syntax for instantiating and applying the `SpikePlasticityRule`.

```
ef = NEFEnsembleFactoryImpl()
pre = ef.make('pre', 100, 1) # 100 neurons, 1 dimension
post = ef.make('post', 200, 1) # 200 neurons, 1 dimension
# Add a termination for modulatory input
post.addDecodedTermination('mod', [[1]], 0.005, True)
# Add a non-decoded termination that will be learned
post.addTermination('stdp', weights, 0.005, False)
error = ef.make('error', 150, 1) # 150 neurons, 1 dimension
inFcn = InSpikeErrorFunction([n.scale for n in post.nodes],post.encoders)
outFcn = OutSpikeErrorFunction([n.scale for n in post.nodes],post.encoders)
rule = SpikePlasticityRule(inFcn, outFcn, 'AXON', 'mod');
post.setPlasticityRule('stdp',rule)
```

If the learning function you wish to use is not implemented in the Java source code, you can easily define it within the same Python script that uses your learning rule. The following example uses the `SimpleSpikeLearningFunction` described in section 3.2.2.

```
class SimpleSpikeLearningFunction(AbstractSpikeLearningFunction)
    def deltaOmega(self, time_since_different, time_since_same, current_weight,
            mod_input, post_index, pre_index, dim):
        return self.myLearningRate * time_since_different

...

ef = NEFEnsembleFactoryImpl()
pre = ef.make('pre', 100, 1) # 100 neurons, 1 dimension
post = ef.make('post', 200, 1) # 200 neurons, 1 dimension
# Add a non-decoded termination that will be learned
post.addTermination('stdp', weights, 0.005, False)
simpleLearnFcn = SimpleSpikeLearningFunction()
rule = SpikePlasticityRule(simpleLearnFcn, simpleLearnFcn, 'AXON', None)
post.setPlasticityRule('stdp',rule)
```

4.3 Using CompositePlasticityRule

In general, it is recommended that you avoid using the `CompositePlasticityRule` class, because it adds additional overhead to an already computationally intensive task. However, if the learning occurring in your system is simple, and you wish to be able to change the simulation mode on the fly, then the `CompositePlasticityRule` is available.

Using the `CompositePlasticityRule` requires knowledge of how to use both `RealPlasticityRule` and `SpikePlasticityRule`, as these are both used in its constructor.

`public CompositePlasticityRule(PlasticityRule spikeRule, PlasticityRule realRule)`
    The `CompositePlasticityRule` constructor requires two `PlasticityRules`, one that will be used when in a spiking simulation mode, and one that will be used when in a rate-based simulation mode.

`CompositePlasticityRule` is a passthrough class for the two plasticity rules it contains. It does not contain any learning logic, it only identifies what simulation mode is being used and calls the methods in the appropriate plasticity rule.

The following is an example using the simple learning functions from sections 3.1.2 and 3.2.2.

```
class SimpleRealLearningFunction(AbstractRealLearningFunction)
    def deltaOmega(self, input, time, current_weight, mod_input, origin_state,
            post_index, pre_index, dim):
        return self.myLearningRate * input * origin_state

class SimpleSpikeLearningFunction(AbstractSpikeLearningFunction)
    def deltaOmega(self, time_since_different, time_since_same, current_weight,
            mod_input, post_index, pre_index, dim):
        return self.myLearningRate * time_since_different

...

ef = NEFEnsembleFactoryImpl()
pre = ef.make('pre', 100, 1) # 100 neurons, 1 dimension
post = ef.make('post', 200, 1) # 200 neurons, 1 dimension
# Add a non-decoded termination that will be learned
post.addTermination('learn', weights, 0.005, False)

realLearnFcn = SimpleRealLearningFunction()
realRule = RealPlasticityRule(realLearnFcn, 'AXON', None)
spikeLearnFcn = SimpleSpikeLearningFunction()
spikeRule = SpikePlasticityRule(spikeLearnFcn, spikeLearnFcn, 'AXON', None)

rule = CompositePlasticityRule(spikeRule, realRule)
post.setPlasticityRule('learn',rule)
```

4.4 Extending plasticity rules

The intention of separating the learning function (the $\Delta\omega_{ij}$) and the plasticity rule (when $\Delta\omega_{ij}$ is applied) was to make it easy to add new learning functions, because most rules are either applied on every timestep or on each spike. Because of this, there should be few situations in which plasticity rules need to be extended.

Some situations where the existing plasticity rules would not be sufficient include the following.

1. The rule requires more than one modulatory input term, or vastly different logic is to occur for different dimensions of the modulatory input.
2. Each connection weight is to decay at a certain rate, independent of the learning that is applied to it.
3. The learning rule is to be applied less often than usual, because applying it too often slows down the simulation too much.
4. The learning rule is used in simulation modes other than rate-based or spiking – i.e. `SimulationMode.APPROXIMATE`, `SimulationMode.DIRECT` and `SimulationMode.PRECISE`.

For the first three cases, there are other alternatives to writing a new plasticity rule.

1. The logic for each dimension can be implement in the learning function, with an `if` or `switch` statement depending on the value of `dim`. If a new plasticity rule is to accept multiple modulatory inputs, it would be preferred that the existing plasticity rules be modified to accept multiple modulatory inputs instead, as long as doing so does not come at a significant speed cost.

2. Decay can easily be implemented in a learning function for rate-based simulations. This may be necessary for spike-based simulations.

3. This is already implemented for rate-based simulations, with the "plasticity interval." See section 5, or the `PlasticEnsemble` interface, for more details.

The simulation modes that are not yet represented by plasticity rules should have rules implemented for them eventually. At the moment, few simulations using plasticity have been done in Nengo, so the focus is on improving how plasticity is implemented for spiking and rate-based simulations, the two most commonly used simulation modes. However, as plasticity becomes more commonly used, these other simulation modes will need to have plasticity rules implemented.

For the cases in which new plasticity rules need to be created, an existing plasticity rule can be subclassed (e.g. for adding weight decay to the `SpikePlasticityRule`), or an existing rule can be used as a template to create a new plasticity rule (e.g. for creating a `DirectPlasticityRule`).

5 Other relevant pieces of code

Developers who are having difficulty fully understanding plasticity rules and when they are applied should examine the following sections of code.

`ca.nengo.model.PlasticEnsemble`
: The `PlasticEnsemble` interface defines the methods that must be implemented by an ensemble that can have plasticity rules applied to it. At the moment, `Ensembles` are the only type of `Node` that can have plasticity rules applied to it. At the moment, there are no plans to change this (this is discussed in appendix A.2).

`ca.nengo.model.PlasticTermination`
: The `PlasticTermination` interface defines the methods that must be implemented by a termination that can be modified by a plasticity rule.

`ca.nengo.model.impl.PlasticEnsembleImpl`
: The `PlasticEnsembleImpl` is a class that implements the `PlasticEnsemble` interface. The two most important methods that deal with plasticity are `addTermination` and `learn`. `addTermination` shows how we ensure that a termination is plastic; `learn` shows how the values from the `getDerivative` function in the plasticity rules are applied.

`ca.nengo.model.impl.PlasticEnsembleTermination`
: The `PlasticEnsembleTermination` is a class that implements the `PlasticTermination` interface. It is essentially a passthrough to a set of `LinearExponentialTerminations`. `LinearExponentialTerminations` are the only node-level terminations that can be used in a plastic ensemble, currently.

A Changes made in SVN revision 985

General information about SVN revision 985 can be found at `http://ctnsrv.uwaterloo.ca:8080/sventon/repos/Nengo/info?revision=985`. The following is a more detailed account of the changes made. All of the actions listed on that page are included.

A.1 Reimplement main plasticity classes

These changes are essentially those discussed in the main text of this report. Before this revision, these classes either didn't exist, or were unmaintained. Those classes that did exist were modified to be easier to use and extend, and many bugs were fixed; `SpikePlasticityRule`, for instance, did not properly update the `outSpiking` array before this revision.

Modified `/simulator/src/java/main/ca/nengo/model/plasticity/PlasticityRule.java`
Made resettable, changed some method names.

Modified `/simulator/src/java/main/ca/nengo/model/plasticity/impl/RealPlasticityRule.java`
Many changes; see `diff` for details.

Modified `/simulator/src/java/main/ca/nengo/model/plasticity/impl/SpikePlasticityRule.java`
Many changes; see `diff` for details.

Modified `/simulator/src/java/test/ca/nengo/model/plasticity/impl/SpikePlasticityRuleTest.java`
Commented out for now. The tests need to be rewritten from scratch.

Modified `/simulator/src/java/main/ca/nengo/model/plasticity/impl/CompositePlasticityRule.java`
Many changes; see `diff` for details.

Added `/simulator/src/java/main/ca/nengo/model/plasticity/impl/AbstractRealLearningFunction.java`

Added `/simulator/src/java/main/ca/nengo/model/plasticity/impl/AbstractSpikeLearningFunction.java`

Added `/simulator/src/java/main/ca/nengo/model/plasticity/impl/ErrorLearningFunction.java`

Added `/simulator/src/java/main/ca/nengo/model/plasticity/impl/InSpikeErrorFunction.java`

Added `/simulator/src/java/main/ca/nengo/model/plasticity/impl/OutSpikeErrorFunction.java`

A.2 Consolidate all plasticity into ensembles

Before this revision, plasticity was implemented both at the ensemble level and the neuron level. While this makes sense conceptually, in that a learning rule that works on an ensemble (one of the form $\Delta\omega_{ij}$) can be adapted to work on a single neuron (essentially one of the form $\Delta\omega_j$), this makes the implementation of plasticity much more difficult. It necessitates the addition of different types of plasticity rules, or creates a general confusion of what should be passed to a plasticity rule, which was the state of Nengo prior to this revision.

It was much easier to consolidate all plasticity implementation details in the ensemble level. Note that this does not lose any functionality; if one wishes to learn on a single neuron, one can create an ensemble with only one neuron, or use a learning function that returns 0 for every $\Delta\omega_{ij,i\neq x}$.

The file-level changes are those listed below. Modified files can be `diff`ed on the website listed above.

Added `/simulator/src/java/main/ca/nengo/model/PlasticEnsemble.java`

Added `/simulator/src/java/main/ca/nengo/model/PlasticTermination.java`

Added `/simulator/src/java/main/ca/nengo/model/impl/PlasticEnsembleImpl.java`

Added `/simulator/src/java/main/ca/nengo/model/impl/PlasticEnsembleTermination.java`

Deleted `/simulator/src/java/main/ca/nengo/model/plasticity/Plastic.java`
`Plastic` interface is no longer needed; only one type of node can be plastic.

Modified `/simulator/src/java/main/ca/nengo/model/nef/NEFEnsemble.java`
Removed implementation of `Plastic` interface (implements `PlasticEnsemble` implicitly through implementing `DecodedEnsemble` instead).

Modified `/simulator/src/java/main/ca/nengo/model/impl/EnsembleImpl.java`
Extracted code that implemented learning in an `EnsembleImpl`, moved to `PlasticEnsembleImpl`.

Modified `/simulator/src/java/main/ca/nengo/model/nef/impl/NEFEnsembleImpl.java`
Extracted code that implemented learning, now availabe due to being a subclass of `PlasticEnsembleImpl`.

Modified `/simulator/src/java/main/ca/nengo/model/impl/EnsembleTermination.java`
Made termination array protected, so that `PlasticEnsembleTermination` termination has access.

Modified `/simulator/src/java/main/ca/nengo/model/neuron/impl/LinearSynapticIntegrator.java`
Removed plasticity logic. No longer a part of neurons or synaptic integrators.

Modified `/simulator/src/java/main/ca/nengo/model/nef/DecodableEnsemble.java`
`DecodableEnsemble` is now a subclass of `PlasticEnsemble` rather than `Ensemble`.

Modified `/simulator/src/java/main/ca/nengo/model/nef/impl/DecodableEnsembleImpl.java`
`DecodableEnsembleImpl` is now a subclass of `PlasticEnsembleImpl` rather than `EnsembleImpl`.

A.2.1 Rename `PlasticExpandableSpikingNeuron` to `ExpandableSpikingNeuron`

As a byproduct of the above consolidation, the `PlasticExpandableSpikingNeuron` – which is the neuron most commonly instantiated when creating new `NEFEnsembles` – had to be renamed to `ExpandableSpikingNeuron`. This change is the most likely to cause compatibility issues with previous versions of Nengo. `.nef` files created before this revision will not load properly, but Python scripts should, for the most part, work without issue.

Below, all of the file modifications were simply replacing all instances of `PlasticExpandableSpikingNeuron` to `ExpandableSpikingNeuron`.

Renamed `/simulator/src/java/main/ca/nengo/model/neuron/impl/PlasticExpandableSpikingNeuron.java`

Modified `/simulator/src/java/main/ca/nengo/model/neuron/impl/ALIFNeuronFactory.java`

Modified `/simulator/src/java/main/ca/nengo/model/neuron/impl/HodgkinHuxleySpikeGenerator.java`

Modified `/simulator/src/java/main/ca/nengo/model/neuron/impl/LIFNeuronFactory.java`

Modified `/simulator/src/java/main/ca/nengo/model/neuron/impl/PoissonSpikeGenerator.java`

Modified `/simulator/src/java/main/ca/nengo/model/neuron/impl/SpikingNeuronFactory.java`

Modified `/simulator/src/java/test/ca/nengo/model/neuron/impl/IzhikevichSpikeGeneratorTest.java`

Modified `/simulator/src/resources/ca/nengo/config/impls.txt`

A.3 Creating interactive plasticity plots

In order to closer examine what is happening when plasticity rules are applied, two graphs were added to Nengo's interactive plots feature. The first is a visualization of the current connection weights of a particular `PlasticEnsembleTermination`. The existing "grid" plot type was modified to allow for both positive and negative values.

The second plot is a simple spike raster with a line graph overlayed, to display the spikes of a presynaptic and postsynapitc neuron, overlayed with the connection weight between them over time. This makes it easy to see the change in weight as a function of the spike patterns of the two neurons. See figure 1, taken from [2].

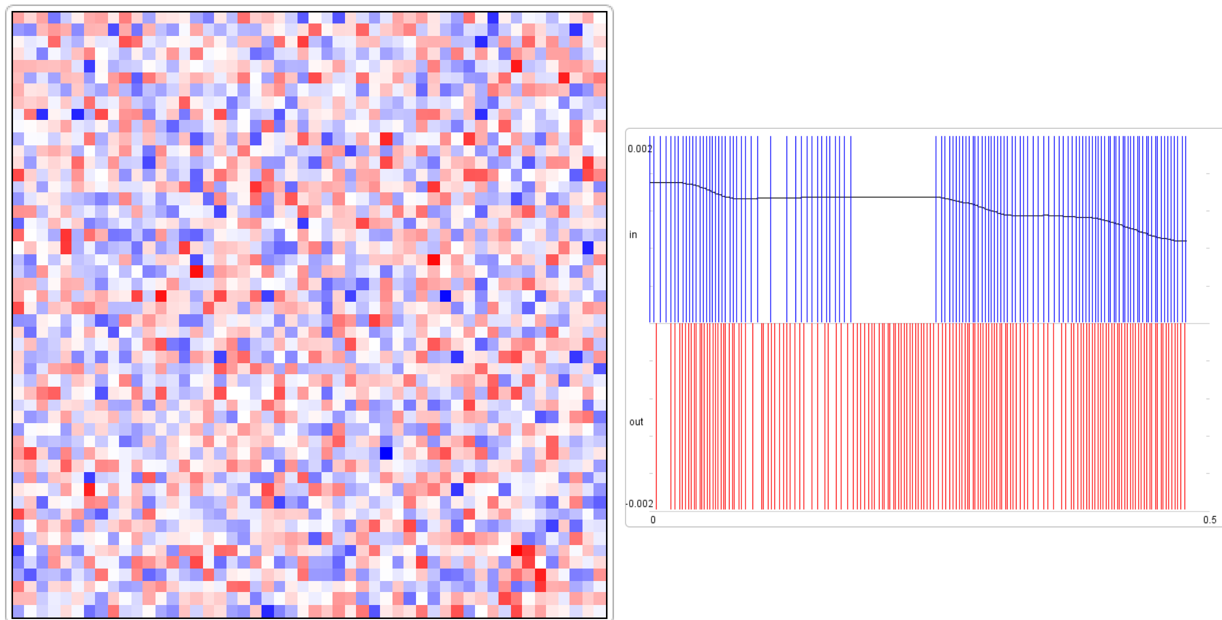

Figure 1: (Left) Visualization of the current connections weights – blue represents a negative weight, red positive. (Right) Visualization of the connection weight between a single pair of neurons, overlaid on a spike raster of the two neurons.

The file-level changes follow.

Modified `/simulator-ui/python/timeview/components/__init__.py`
Added `spike_line_overlay` to the list of available components.

Modified `/simulator-ui/python/timeview/components/grid.py`
Allowed for positive and negative values (blue for negative, red for positive).

Added `/simulator-ui/python/timeview/components/spike_line_overlay.py`
Adds a plot that is a spike raster with a line graph overlayed.

Modified `/simulator-ui/python/timeview/view.py`
Adds GUI elements to access the connection weight plots.

A.4 Make learning resettable

Previously, if the connection weights of a termination were modified by a simulation with a learning rule, they would remain modified if the simulation was run a second time. That is, once the weights on a termination were learned, they were permanently changed. This was fixed with the following changes.

Modified `/simulator/src/java/main/ca/nengo/model/impl/LinearExponentialTermination.java`
Added a `reset` method to restore termination weights to their original values.

Modified `/simulator/src/java/main/ca/nengo/model/nef/impl/DecodedTermination.java`
Added a `reset` method to restore termination weights to their original values.
Note: A reset method should not have been added to DecodedTermination, because its weights are mutable through the GUI and the scripting interface. Changes to a DecodedTermination's weights are designed to persist. This was fixed in SVN revision 998.

A.5 Bugfix: ensure mode is correct on first timestep

In implementing the plasticity rules, a bug was encountered in which a spiking simulation would emit `RealOutput` on the first timestep. This was because the spike generator object was put into constant rate mode when the simulation began, in order to determine the tuning curves; once the rate mode was switched to the default spiking mode, it would only update on the second timestep. This caused a problem with the `SpikePlasticityRule` because it explicitly checks for `SpikeOutput` on each timestep.

Modified `/simulator/src/java/main/ca/nengo/model/neuron/impl/SpikeGeneratorOrigin.java`
Added a `setMode` function to generate the appropriate output type.

Modified `/simulator/src/java/main/ca/nengo/model/neuron/impl/SpikingNeuron.java`
Calls the spike generator's `setMode` function.

References

[1] Larry F. Abbott and Sacha B. Nelson. Synaptic plasticity: taming the beast. *Nature Neuroscience*, 3:1178–83, November 2000.

[2] Trevor Bekolay. A general error-based spike-timing dependent learning rule for the Neural Engineering Framework, 2010.

[3] Guo-Qiang Bi and Mu-Ming Poo. Synaptic modification by correlated activity: Hebb's postulate revisited. *Annual Review of Neuroscience*, 24(1):139–66, January 2001.

[4] Elie L. Bienenstock, Leon N. Cooper, and Paul Munro. Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex. *Journal of Neuroscience*, 2(1):32, 1982.

[5] Natalia Caporale and Yang Dan. Spike timing-dependent plasticity: a Hebbian learning rule. *Annual Review of Neuroscience*, 31(1):25–46, 2008.

[6] Chris Eliasmith and Charles H. Anderson. *Neural engineering: computation, representation, and dynamics in neurobiological systems*. Computational neuroscience. MIT Press, 2003.

[7] David Macneil and Chris Eliasmith. Fine-tuning and stability of recurrent neural networks. *Neuron (Preprint)*, 2010.

[8] S. J. Martin, P. D. Grimwood, and R. G. Morris. Synaptic plasticity and memory: an evaluation of the hypothesis. *Annual Review of Neuroscience*, 23:649–711, January 2000.

[9] Guilherme Neves, Sam F. Cooke, and Tim V. P. Bliss. Synaptic plasticity, memory and the hippocampus: a neural network approach to causality. *Nature Reviews Neuroscience*, 9(1):65–75, January 2008.