Chapter 15 Neural Plasticity

Abstract In this chapter we add plasticity to our direct encoded NN system. We implement numerous plasticity encoding approaches, and develop numerous plasticity learning rules, amongst which are variations of the Hebbian Learning Rule, Oja's Rule, and Neural Modulation. Once plasticity has been added, we again test our TWEANN system on the T-Maze navigation benchmark.

We have now built a truly advanced topology and weight evolving artificial neural network (TWEANN) platform. Our system allows for its various features to evolve, the NNs can evolve not only the topology and synaptic weights, but also evolutionary strategies, local and global search parameters, and the very way in which the neurons/processing-elements interact with input signals. We have implemented our system in such a way that it can easily be further expanded and extended with new activation functions (such as logical operators, or activation functions which simulate a transistor for example), mutation operators, mutation strategies, and almost every other feature of our TWEANN. We have also created two benchmarks, the double pole balancing benchmark and the T-Maze navigation benchmark, which allows us to test our system's performance.

There is something lacking at this point though, our evolved agents are but static systems. Our NN based agents do not learn during their lifetimes, they are trained by the exoself, which applies the NN based system to the problem time after time, with different parameters, until one of the parameter/synaptic-weight combinations produces a more fit agent. This is not learning. Learning is the process during which the NN changes due to its experience, due to its interaction with the environment. In biological organisms, evolution produces the combination of neural topology, plasticity parameters, and the starting synaptic weight values, which allows the NN, based on this plasticity and initial NN topology and setup, to learn how to interact with the environment, to learn and change and adapt during its lifetime. The plasticity parameters allow the NN to change as it interacts with the environment. While the initial synaptic weight values send this newborn agent in the right direction, in hope that the plasticity will change the topology and synaptic weights in the direction that will drive the agent, the organism, further in its exploration, learning, adaptation, and thus towards a higher fitness.

Of course with plasticity comes a new set of questions: What new mutation operators need to be added? How do we make the mutation operators specific to that particular set of parameters used by the plasticity learning rule? What about the tuning phase when it comes to neurons with plasticity, what is the difference between plasticity enabled NNs which are evolved through genetic algorithm approaches, and those evolved through memetic algorithm approaches? During the

G.I. Sher, *Handbook of Neuroevolution Through Erlang*,
DOI 10.1007/978-1-4614-4463-3_15,

tuning phase, what do we perturb, the synaptic weights or the plasticity parameters?...

Plasticity is that feature which allows the neuron and its parameters to change due to its interaction with input signals. In this book's neural network foundations chapters we discussed this in detail. In this chapter we will implement the various learning rules that add neural plasticity to our system. In this chapter we will create 3 types of plasticity functions, the standard Hebbian plasticity, the more advanced Oja's rule, and finally the most dynamic and flexible approach, neural plasticity through neuromodulation. We will first discuss and implement these learning rules, and then add the perturbation and mutation operators necessary to take advantage of the newly added learning mechanism.

15.1 Hebbian Rule

We discussed the Hebbian learning rule in Section-2.6.1. The principle behind the Hebbian learning rule is summarized by the quote "Neurons that fire together, wire together." If a presynaptic neuron A which is connected to a neuron B, sends it an excitatory (*SignalVal > 0*) signal, and in return B produces an excitatory output, then the synaptic weight between the two neurons increases in magnitude. If on the other hand neuron A sends an excitatory signal to B, and B's resulting output signal is inhibitory (*SignalVal < 0*), then B's synaptic weight for A's connection, decreases. In a symmetric fashion, an inhibitory signal from A that results in an inhibitory signal from B, increases the synaptic weight strength between the two, but an inhibitory signal from A resulting in an excitatory signal from B, decreases the strength of the connection.

The simplest Hebbian rule used to modify the synaptic weight after the neuron has processed some signal at time *t* is:

$$Delta\_Weight = h * I\_Val * Output,$$

Thus:

$$W_{(t+1)} = W_{(t)} + Delta\_Weight.$$

Where Delta_Weight is the change in the synaptic weight, and where the specified synaptic weight belongs to B, associated with the incoming input signal *I_Val*, coming from neuron A. The value *h* is the learning parameter, set by the researcher. The algorithm and architecture of a neuron using a simple Hebbian learning rule, repeated from Section-2.6.1 for clarity, is shown in Fig-15.1.

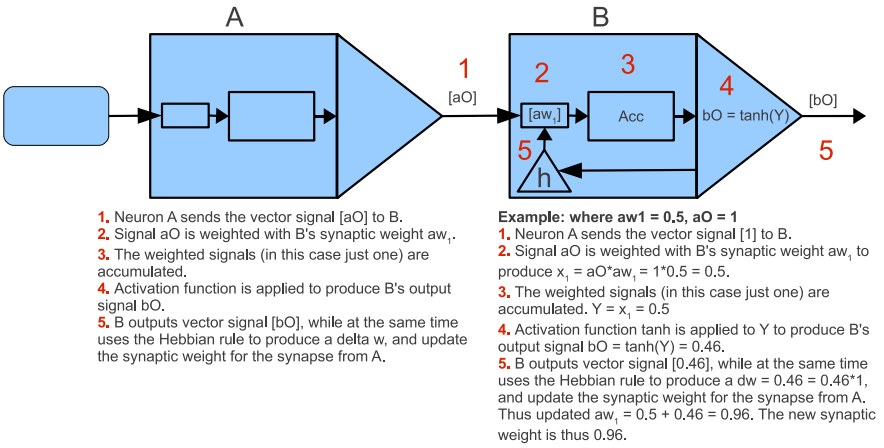


Fig. 15.1 An architecture of a neuron using the Hebbian learning rule based plasticity.

This is the simplest Hebbian rule, but though computationally light, it is also unstable. Because the synaptic weight does not decay, if left unchecked, the Hebbian rule will keep increasing the magnitude of the synaptic weight, indefinitely, and thus eventually drown out all other synaptic weights belonging to the neuron. For example, if a neuron has 5 synaptic weights, 4 of which are between -Pi and P, and the fifth weight has climbed to 1000, this neuron is effectively useless with regards to processing since the signal weighted by 1000 will most likely overpower other signals. No matter what the other 4 synaptic weights are, no matter what pattern they have evolved to pick up, the fifth weight with magnitude 1000 will drown out everything, saturating the output. We will implement it for the sake of completeness, and also because it is so easy to implement. To deal with unchecked synaptic weight magnitudes, we will use our previously created *functions:sat/1* and *functions:sat/2* functions to ensure that the synaptic weights do not increase in magnitude unchecked, that they do not increase to infinity, and instead get saturated at some level specified by the *sat* function and the *?SAT_LIMIT* parameter specified within the neuron module.

There is though a problem with the current architecture of our neuron, which prevents it from having plasticity. That problem is that the neuron's *input_idps* list specifies only the *Input_Id* of the node that sends it an input signal, and the accompanying synaptic weight list *Weights*: *[{Input_Id,Weights}...]*. With the addition of plasticity, we must have the ability to also specify the various new parameters (like the learning parameter for example) of the learning rule. There are multiple ways in which we can solve this dilemma, the following are four of them:

1. Extend the input_idps from: *[{Input_Id,Weights}...]* to: *[{Input_Id, Weights, LearningParameters}...]*
2. Extend the *neuron* record to also include *input_lpps*, a list with the format: *[{Input_Id,LPs}...]*, where *input_lpps* stands for *input learning parameters*

plus, and the *LPs* list in the tuple stands for *Learning Parameters*, mirroring the *input_idps* list's format.

3. Extend the *Weights* list in the *input_idps* tuple list from: $[W_1,W_2,W_3...]$ To: $[\{W_1,P_1\},\{W_2,P_2\},\{W_3,P_3\}...]$
4. Extend *pf* (Plasticity Function) specification from: *atom()::FunctionName* to: *{atom()::FunctionName, ParameterList}*

All of these solutions would require us to modify the *genotype, genome_mutator, exoself, neuron, signal_aggregator*, and *plasticity* modules, so that these modules can properly create, mutate, map genotype to phenotype, and in general properly function when the NN system is active. DXNN uses the 3rd solution, but only because at one point I also allowed the evolved NN systems to use a modified back propagation learning algorithm, and P_i contained the learning parameter. There were also D_i and M_i parameters, making the *input_idps* list of the neurons evolved by the DXNN platform have the following format: $[\{W_1,P_1,D_1,M_1\},\{W_2,P_2,D_2,M_2\}...]$, where the value D contained the previous time step's change in synaptic weight, and M contained the momentum parameter used by the backprop algorithm.

Options 1-3 are appropriate for when there is a separate plasticity function, a separate synaptic weight modification and learning rule, for every synaptic weight. But in a lot of cases, the neuron has a single learning rule which is applied to all synaptic weights equally. This is the case with the Hebbian Learning Rule, where the neuron needs only a single learning parameter specifying the rate of change of the synaptic weights. For the learning rules that use a single parameter or a list of global learning parameters, rather than a separate list of learning parameters for every synaptic weight, option 4 is the most appropriate, in which we extend the plasticity function name with a parameter list used by that plasticity function.

But what if at some point in the future we decide that every weight should be accompanied not by one extra parameter, but by 2, or 3, or 4... To solve this, we could use solution-3, but have each P_i be a list. If there is only one parameter, then it is a list of length 1: $[A_1]$, if two parameters are needed by some specific learning rule, then each P is a list of length 2: $[A_1,A_2]$, and so on. If there is no plasticity, the list is empty.

Are there such learning rules that require so many parameters? Yes, for example some versions of neuromodulation can be set such that a single neuron simulates having 5 other modulating neurons within, each of whom analyzes the input vectors to the neuron in question, and each of whom outputs a value which specifies a particular parameter in the generalized Hebbian learning rule. This type of plasticity function could use anywhere from 2 to 5 parameters (in the version we will implement) for each synaptic weight (those 2-5 parameters are themselves synaptic weights of the *embedded* modulating neurons), and we will discuss that particular approach and neuromodulation in general in section 15.3. Whatever rule we choose, there is a price. Luckily though, due to the way we've constructed our

system, it is easy to fix and modify it, no matter which of the listed approaches we decide to go with.

Let us choose the 3rd option where each P_i is a list of parameters for each weight W_i, and where that list length is dependent on the plasticity function the neuron uses. In addition, we will also implement the 4th option, which requires us to modify the *pf* parameter format. The pf parameter for every neuron will be specified as a tuple, composed of the plasticity function name and a global learning parameter list. This will, though making the implementation a bit more difficult, allow for a much greater level of flexibility in the types of plasticity rules we can implement. Using both methods, we will have access to plasticity functions which need to specify a parameter for every synaptic weight, and those which only need to specify a single or a few global parameters of the learning rule for the entire neuron.

15.1.1 Implementing the New input_idps & pf Formats

We first update the specification format for the neuron's pf parameter. This requires only a slight modification in the neuron module, changing the line:

```
U_IPIdPs =plasticity:PF(Ordered_IAcc,Input_PIdPs,Output)
```

To:

```
{PFName,PFParameters} = PF,
U_IPIdPs = plasticity:PFName(PFParameters,Ordered_IAcc,Input_PIdPs,Output),
```

And a change in the genotype module, to allow us to use the plasticity function name to generate the *PF* tuple. The way we do this is by creating a special function in the plasticity module with arity 1 and of the form: *plasticity:PFName(neural_parameters)*, which returns the necessary plasticity function specifying tuple: *{PFName, PL}*, where PL is the Parameter List. In this manner, when we develop the plasticity functions, we can at the same time create the function of arity 1 which returns the appropriate tuple defining the actual plasticity function name and its parameters. The change in the genotype module is done to the generate_NeuronPF/1 function, changing it from:

```
generate_NeuronPF(Plasticity_Functions)->
    case Plasticity_Functions of
        [] ->
            none;
        Other ->
            lists:nth(random:uniform(length(Other)),Other)
```

```
    end.
```

To:

```
generate_NeuronPF(Plasticity_Functions)->
    case Plasticity_Functions of
        [] ->
            {none,[]};
        Other ->
            PFName = lists:nth(random:uniform(length(Other)),Other),
            plasticity:PFName(neural_parameters)
    end.
```

With this modification completed, we can specify the global, neural level learning parameters. But to be able to specify synaptic weight level parameters, we have to augment the neuron's input_idps list specification format. Because our new format for input_idps stays very similar to the original, we need only convert the original list's form from: *[{Input_Id, Weights}...]* to: *[{Input_Id,WeightsP}...]*. Any function that does not directly operate on Weights, does not get affected by us changing Weights: $[W_1,W_2...]$ to WeightsP: $[\{W_1,PL_1\},\{W_2,PL_2\}...]$, where PL is the plasticity function's Parameter List. The only function that does get affected by this change is the one in the genotype module which creates the *input_idps* list, *create_NeuralWeights/2*. In *genome_mutator* module, again the only affected function is the *mutate_weights* function which uses the *perturb_weights* function and thus needs to choose the weights rather than the learning parameters to mutate. Finally, the *neuron* process also perturbs its synaptic weights, and so we will need to use a modified version of the *perturb_weights* function.

The most interesting modification occurs in the *create_NeuralWeights* function. We modify it from:

```
create_NeuralWeights(0,Acc) ->
    Acc;
create_NeuralWeights(Index,Acc) ->
    W = random:uniform()-0.5,
    create_NeuralWeights(Index-1,[W|Acc]).
```

To:

```
create_NeuralWeightsP(_PFName,0,Acc) ->
    Acc;
create_NeuralWeightsP(PFName,Index,Acc) ->
    W = random:uniform()-0.5,
    create_NeuralWeightsP(PFName,Index-1,[{W,plasticity:PFName(weight_parameters)} |
Acc]).
```

The second version creates a list of tuples rather than a simple list of synaptic weights. Since each learning rule, each plasticity function, will have its own set of parameters, we defer the creation of a parameter list to its own plasticity function. To have the plasticity function create an initial synaptic level parameter list, we will call it with the atom parameter: *weight_parameters*. Thus for every plasticity function, we will create a secondary clause, which takes as input a single parameter, and through the use of this parameter it will specify whether the plasticity function will return neural level learning rule parameters, or synaptic weight level learning rule parameters. The *weight_parameters* specification will make the plasticity function return a randomized list of parameters required by that learning rule at the synaptic weight level.

We also add to the plasticity module a secondary *none* function: none/1. This none/1 function can be executed with the *neural_parameters* or the *weight_parameters* atom, and in both cases it returns an empty list, since a neuron which does not have plasticity and thus uses the *none/1* plasticity function, does not need learning parameters of any type. Thus, our plasticity module now holds two functions by the name *none:* one with arity 4, and one with arity 1:

```
none(neural_parameters)->
    [];
none(weight_parameters)->
    [].
%none/0 returns a set of learning parameters needed by the none/0 plasticity function. Since
this function specifies that the neuron has no plasticity, the parameter lists are empty.

none(_NeuralParameters,_IAcc,Input_PIdPs,_Output)->
    Input_PIdPs.
%none/3 returns the original Input_PIdPs to the caller.
```

The modification to the *perturb_weights* function (present in the neuron module, and present in the genome_mutator module in a slightly modified form) is much simpler. The updated function has the form, where the changes have been highlighted in boldface:

```
perturb_weights(Spread,MP,[{W,LPs}|Weights],Acc)->
    U_W = case random:uniform() < MP of
            true->
                sat((random:uniform()-0.5)*2*Spread+W,-?SAT_LIMIT,?SAT_LIMIT);
            false ->
                W
    end,
    perturb_weightsP(Spread,MP,Weights,[{U_W,LPs}|Acc]);
perturb_weightsP(_Spread,_MP,[],Acc)->
    lists:reverse(Acc).
```

All that has changed is the function name, and that instead of using: [W|Weights], we now use: [{W,LPs}|WeightsP], where the list LPs stands for Learning Parameters.

Finally, we must also update the synaptic weight and plasticity function specific mutation operators. These functions are located in the genome_mutator module. These are the *add_bias/1*, *mutate_pf/1*, and the *link_ToNeuron/4* functions. The add_bias/1 and link_ToNeuron/4 functions add new synaptic weights, and thus must utilize the new *plasticity:PFName(weight_parameters)* function, based on the particular plasticity function used by the neuron. The mutate_pf/1 is a mutation operator function. Due to the extra parameter added to the input_idps list, when we mutate the plasticity function, we must also update the synaptic weight parameters so that they are appropriate for the format of the new learning rule. Only the mutate_pf/1 function requires a more involved modification to the source code, with the other two only needing for the plasticity function name to be extracted and used to generate the weight parameters from the plasticity module. The updated mutate_pf/1 function is shown in Listing-15.1, with the modified parts in boldface.

Listing-15.1 The updated implementation of the mutate_pf/1 function.

```
mutate_pf(Agent_Id)->
    A = genotype:read({agent,Agent_Id}),
    Cx_Id = A#agent.cx_id,
    Cx = genotype:read({cortex,Cx_Id}),
    N_Ids = Cx#cortex.neuron_ids,
    N_Id = lists:nth(random:uniform(length(N_Ids)),N_Ids),
    Generation = A#agent.generation,
    N = genotype:read({neuron,N_Id}),
    {PFName,_NLParameters} = N#neuron.pf,
    case (A#agent.constraint)#constraint.neural_pfns -- [PFName] of
            [] ->
                    exit("********ERROR:mutate_pf:: There are no other plasticity functions to
use.");
            Other_PFNames ->

New_PFName=lists:nth(random:uniform(length(Other_PFNames)),Other_PFNames),
                    New_NLParameters = plasticity:New_PFName(neural_parameters),
                    NewPF = {New_PFName,New_NLParameters},
                    InputIdPs = N#neuron.input_idps,
                    U_InputIdPs = [{Input_IdP,plasticity:New_PFName(weight_parameters)}
|| {Input_IdP,_OldPL} <- InputIdPs],
                    U_N = N#neuron{pf=NewPF,input_idps = U_InputIdPs, generation
=Generation},
                    EvoHist = A#agent.evo_hist,
```

```
                U_EvoHist = [{mutate_pf,N_Id}|EvoHist],
                U_A = A#agent{evo_hist=U_EvoHist},
                genotype:write(U_N),
                genotype:write(U_A)
end.
```

After making these modifications, we ensure that everything is functioning as it should, by executing:

```
polis:sync().
polis:start().
population_monitor:test().
```

Which compiles the updated modules ensuring that there are no errors, then starts the polis process, and then finally runs a quick neuroevolutionary test. The function *population_monitor:test/0* can be executed a few times (each execution done after the previous one runs to completion), to ensure that everything still works. Because neuroevolutionary systems function stochastically, the genotypes and topologies evolved during one evolutionary run will be different from another, and so it is always a good idea to run it a few times, to test out the various combinations and permutations of the evolving agents.

With this update completed, we can now create plasticity functions. Using our plasticity module implementation, we allow the plasticity functions to completely isolate and decouple their functionality and setup from the rest of the system, which will allow others to add and test new plasticity functions as they please, without disturbing or having to dig through the rest of the code.

15.1.2 Implementing the Simple Hebbian Learning Rule

We need to implement a rule where every synaptic weight W_i is updated every time the neuron processes an input vector and produces an output vector. The weight W_i must be updated using the rule: $Updated\_W_i = W_i + h*I_i*Output$, where I_i is the float() input value associated with the synaptic weight W_i. The Updated_W_i must be, in the same way as done during weight perturbation, saturated at the value: *?SAT_LIMIT*, so that its magnitude does not increase indefinitely.

From the above equation, it can be seen from the common h for all I_i and W_i, that the standard Hebbian learning rule is one where the neuron has a single, global, neural level learning parameter *h*, which is used to update all the synaptic weights belonging to that neuron. Because our neuron also has the ability to have a learning parameter per weight, we can also create a Hebbian learning rule where every synaptic weight uses its very own h. Though note that this approach will double the number of mutatable parameters for the neuron: a list of synaptic

weights, and a list of the same size of Hebbian learning parameters. For the sake of completeness, we will implement both versions. We will call the standard Hebbian learning function which uses a single learning parameter *h* for all synaptic weights, *hebbian/4*, and one which uses a separate learning parameter h_i for every synaptic weight, *hebbian_w/4* (where *_w* stands for weights). Let us first implement the *hebbian_w* function, which uses the following weight update rule: $Updated\_W_i = W_i + h_i * I_i * Output$, where W_i is the synaptic weight, h_i is the learning parameter for neuron W_i, and I_i is the input signal associated with synaptic weight W_i.

In the previous section we have updated our neuron to apply a learning rule to its weights through: *U_IPIdPs = plasticity:PFName(Neural_Parameters, Ordered_IAcc,Input_PIdPs,Output),* which gives the plasticity function access to the neural parameters list, the output signal, the synaptic weights and their associated learning parameters, and the accumulated input vector. To set up the plasticity function by the name hebbian_w, we first implement the function hebbian_w/1 which returns a weight parameters list composed of a single element *[H]* when hebbian_w/1 is executed with the *weights_parameters* parameter, and an empty list when it is executed with the neural_parameters parameter. We then create the function hebbian_w/4 which implements this actual learning rule. The implementation of these two *hebbian_w* functions is shown in Listing-15.2.

Listing 15.2 The implementation of hebbian_w/1 and hebbian_w/4 functions.

```
hebbian_w(neural_parameters)->
   [];
hebbian_w(weight_parameters)->
   [(lists:random()-0.5)].
%hebbian_w/1 function produces the necessary parameter list for the hebbian_w learning rule to operate. The weights parameter list generated by hebbian_w learning rule is a list composed of a single parameter H: [H], for every synaptic weight of the neuron. When hebbian_w/1 is called with the parameter neural_parameters, it returns [].

hebbian_w(_NeuralParameters,IAcc,Input_PIdPs,Output)->
   hebbian_w1(IAcc,Input_PIdPs,Output,[]).

   hebbian_w1([{IPId,Is}|IAcc],[{IPId,WPs}|Input_PIdPs],Output,Acc)->
         Updated_WPs = hebbrule_w(Is,WPs,Output,[]),
         hebbian_w1(IAcc,Input_PIdPs,Output,[{IPId,Updated_WPs}|Acc]);
   hebbian_w1([],[],_Output,Acc)->
         lists:reverse(Acc);
   hebbian_w1([],[{bias,WPs}],Output,Acc)->
         lists:reverse([{bias,WPs}|Acc]).
```

```
%hebbian_w/4 function operates on each Input_PIdP, calling the hebbian_w1/4 function which
processes each of the complementary Is and WPs lists, producing the Updated_WPs lists in re-
turn, with the now updated/adapted weights, based on the hebbian_w learning rule.

   hebbrule_w([I|Is],[{W,[H]}|WPs],Output,Acc)->
         Updated_W = functions:saturation(W + H*I*Output,?SAT_LIMIT),
         hebbrule_w(Is,WPs,Output,[{Updated_W,[H]}|Acc]);
   hebbrule_w([],[],_Output,Acc)->
         lists:reverse(Acc).
%hebbrule_w/4 applies the Hebbian learning rule to each synaptic weight by using the input
value I, the neuron's calculated Output, and each W's own distinct learning parameter H.
```

The function hebbian_w/4 calls hebbian_w1/4 with a list accumulator, which separately operates on the input vectors from each Input_PId by calling the hebbrule_w/4 function. It is the hebbrule_w/4 function that actually executes the modified Hebbian learning rule: *Updated_W = functions:saturation(W+H*I*Output, ?SAT_LIMIT)*, and updates the WeightsP list.

Note that hebbian_w/1 generates a parameter list composed of a single value with a range between -0.5 and 0.5 (This range was chosen to ensure that from the very start the learning parameter will not be too large). The Hebbian rule which uses a negative learning parameter embodies Anti-Hebbian learning. The Anti-Hebbian learning rule decreases the postsynaptic weight between neurons outputting signals of the same sign, and increases magnitude of the postsynaptic weight between those neurons that are connected and output signals of differing signs. Thus, if a neuron A sends a signal to neuron B, and the presynaptic signal is positive, while the postsynaptic neuron B's output signal is negative, and it has $H < 0$, and is thus using the Anti-Hebbian learning rule, then the B's synaptic weight for the link from neuron A will increase in magnitude. This means that in the hebbian_w/4 learning rule implementation, some of the synaptic weights will be using Hebbian learning, and some Anti-Hebbian. This will add some extra agility to our system that might prove useful, and allow the system to evolve more general learning networks.

With the modified Hebbian rule now implemented, let us implement the standard one. In the standard Hebbian rule, the hebbian/1 function generates an empty list when called with *weight_parameters*, and the list [H] when called with neural_parameters. Also, the hebbian/4 function that implements the actual learning rule will use a single common *H* learning parameter to update all the synaptic weights in the input_idps. Listing-15.3 shows the implementation of such standard Hebbian learning rule.

Listing-15.3 The implementation of the standard Hebbian learning rule.

```
hebbian(neural_parameters)->
   [(lists:random()-0.5)];
hebbian(weight_parameters)->
```

```
    [].
%The hebbian/1 function produces the necessary parameter list for the Hebbian learning rule to operate. The parameter list for the standard Hebbian learning rule is a list composed of a single parameter H: [H], used by the neuron for all its synaptic weights. When hebbian/1 is called with the parameter weight_parameters, it returns [].

hebbian([H],IAcc,Input_PIdPs,Output)->
    hebbian(H,IAcc,Input_PIdPs,Output,[]).

    hebbian(H,[{IPId,Is}|IAcc],[{IPId,WPs}|Input_PIdPs],Output,Acc)->
            Updated_WPs = hebbrule(H,Is,WPs,Output,[]),
            hebbian(H,IAcc,Input_PIdPs,Output,[{IPId,Updated_WPs}|Acc]);
    hebbian(_H,[],[],_Output,Acc)->
            lists:reverse(Acc);
    hebbian(_H,[],[{bias,WPs}],Output,Acc)->
            lists:reverse([{bias,WPs}|Acc]).
%hebbian/4 function operates on each Input_PIdP, calling the hebbian/5 function which processes each of the complementary Is and WPs lists, producing the Updated_WPs list in return, with the updated/adapted weights based on the standard Hebbian learning rule, using the neuron's single learning parameter H.

    hebbrule(H,[I|Is],[{W,[]}|WPs],Output,Acc)->
            Updated_W = functions:saturation(W + H*I*Output,?SAT_LIMIT),
            hebbrule(H,Is,WPs,Output,[{Updated_W,[]}|Acc]);
    hebbrule(H,[],[],_Output,Acc)->
            lists:reverse(Acc).
%hebbrule/5 applies the Hebbian learning rule to each weight, using the input value I, the neuron's calculated output Output, and the neuron's single learning parameter H.
```

The standard Hebbian learning rule has a number of flaws. One of these flaws is that without the *saturation/2* function that we're using, the synaptic weight would grow in magnitude to infinity. A more biologically faithful implementation of this auto-associative learning, is the Oja's learning rule, which we discuss and implement next.

15.2 Oja's Rule

The Oja's learning rule is a modification of the standard Hebbian learning rule that solves its stability problems through the use of multiplicative normalization, derived in [1]. This learning rule is also closer to what occurs in biological neurons. The synaptic weight update algorithm embodied by the Oja's learning rule is as follows: $Updated\_W_i = W_i + h*O*(I_i - O*W_i)$, where h is the learning parameter, O is the output of the neuron based on its processing of the input vectors using

its synaptic weights, I_i is the i^{th} input signal, and W_i is the i^{th} synaptic weight associated with the I_i input signal.

We can compare the instability of the Hebbian rule to the stability of the Oja's rule by running this learning rule through a few iterations with a positive input signal *I.* Assuming our neuron only has a single synaptic weight for an input vector of length one, we test the stability of the synaptic weight updated through the Oja's rule as follows:

Initial setup: W = 0.5, h = 0.2, activation function is tanh, using a constant input I = 1:

1. O=math:tanh(W*I)=math:tanh(0.5*1)=0.46
   Updated_W = W + h*O*(I – O*W) = 0.5 + 0.2*0.46*(1 – 0.46*0.5) = 0.57
2. O=math:tanh(W*I)=math:tanh(0.57*1)=0.52
   Updated_W = W + h*O*(I – O*W) = 0.57 + 0.2*0.52(1 – 0.52*0.57) = 0.64
3. O=math:tanh(W*I)=math:tanh(0.64*1)=0.56
   Updated_W = W + h*O*(I - O*W) = 0.64 + 0.2*0.56*(1 - 0.56*0.64) = 0.71
4. …

This continues to increase, but once the synaptic weight achieves a value higher than the input, for example when W = 1.5, the learning rule takes the weight update in the other direction:

5. O=math:tanh(W*I)=math:tanh(1.5*1)=0.90
   Updated_W = **W** + h*O*(I - O*W) = **1.5** + 0.2*0.90*(1 - 0.90*1.5) = **1.43**

Thus this learning rule is indeed self stabilizing, the synaptic weights will not continue to increase in magnitude towards infinity, as was the case with the Hebbian learning rule. Let us now implement the two functions, one which returns the needed learning parameters for this learning rule, and the other implementing the actual Oja's synaptic weight update rule.

15.2.1 Implementing the Oja's Learning Rule

Like the Hebbian learning rule, the standard Oja's rule too only uses a single parameter h to pace the learning rate of the synaptic weights. We implement ojas_w/1 in the same fashion we did the hebbian_w/1, it will be a variation of the Oja's learning rule that uses a single learning parameter per synaptic weight, rather than a single learning parameter for the entire neuron. This synaptic weight update rule is as follows:

$$Updated\_W_i = W_i + h_i*O*(I_i – O*W_i)$$

We set the initial learning parameter to be randomly chosen between -0.5 and 0.5. The implementation of ojas_w/1 and ojas_w/4 is shown in Listing-15.4.

Listing-15.4 The implementation of a modified Oja's learning rule, and its initial learning parameter generating function.

```
ojas_w(neural_parameters)->
    [];
ojas_w(synaptic_parameters)->
    [(lists:random()-0.5)].
%oja/1 function produces the necessary parameter list for the Oja's learning rule to operate. The parameter list for Oja's learning rule is a list composed of a single parameter H: [H] per synaptic weight. If the learning parameter is positive, then the postsynaptic neuron's synaptic weight increases if the two connected neurons produce output signals of the same sign. If the learning parameter is negative, and the two connected neurons produce output signals of the same sign, then the synaptic weight of the postsynaptic neuron, decreases in magnitude.

ojas_w(_Neural_Parameters,IAcc,Input_PIdPs,Output)->
    ojas_w1(IAcc,Input_PIdPs,Output,[]).
ojas_w1([{IPId,Is}|IAcc],[{IPId,WPs}|Input_PIdPs],Output,Acc)->
    Updated_WPs = ojas_rule_w(Is,WPs,Output,[]),
    ojas_w1(IAcc,Input_PIdPs,Output,[{IPId,Updated_WPs}|Acc]);
ojas_w1([],[],_Output,Acc)->
    lists:reverse(Acc);
ojas_w1([],[{bias,WPs}],Output,Acc)->
    lists:reverse([{bias,WPs}|Acc]).
%ojas_w/4 function operates on each Input_PIdP, calling the ojas_rule_w/4 function which processes each of the complementary Is and WPs lists, producing the Updated_WPs list in return. In the returned Updated_WPs, the updated/adapted weights are based on the oja's learning rule, using each synaptic weight's distinct learning parameter.

    ojas_rule_w([I|Is],[{W,[H]}|WPs],Output,Acc)->
            Updated_W = functions:saturation(W + H*Output*(I - Output*W),?SAT_LIMIT),
            ojas_rule_w(Is,WPs,Output,[{Updated_W,[H]}|Acc]);
    ojas_rule_w([],[],_Output,Acc)->
            lists:reverse(Acc).
%ojas_weights/4 applies the oja's learning rule to each weight, using the input value I, the neuron's calculated output Output, and each weight's distinct learning parameter H.
```

The standard implementation of Oja's learning rule, which uses a single learning parameter H for all synaptic weights, is shown in Listing-15.5. The standard Oja's rule uses the following weight update algorithm: $Updated\_W_i = W_i + h*O*(I_i - O*W_i)$.

Listing-15.5 The implementation of the standard Oja's learning rule.

```
ojas(neural_parameters)->
    [(lists:random()-0.5)];
```

```
ojas(synaptic_parameters)->
    [].
%oja/1 function produces the necessary parameter list for the oja's learning rule to operate. The
parameter list for oja's learning rule is a list composed of a single parameter H: [H], used by the
neuron for all its synaptic weights. If the learning parameter is positive, and the two connected
neurons produce output signals of the same sign, then the postsynaptic neuron's synaptic
weight increases. Otherwise it decreases.

ojas([H],IAcc,Input_PIdPs,Output)->
    ojas(H,IAcc,Input_PIdPs,Output,[]).
ojas(H,[{IPId,Is}|IAcc],[{IPId,WPs}|Input_PIdPs],Output,Acc)->
    Updated_WPs = ojas_rule(H,Is,WPs,Output,[]),
    ojas(H,IAcc,Input_PIdPs,Output,[{IPId,Updated_WPs}|Acc]);
ojas(_H,[],[],_Output,Acc)->
    lists:reverse(Acc);
ojas(_H,[],[{bias,WPs}],Output,Acc)->
    lists:reverse([{bias,WPs}|Acc]).
%ojas/5 function operates on each Input_PIdP, calling the ojas_rule/5 function which processes
each of the complementary Is and WPs lists, producing the Updated_WPs list in return, with the
updated/adapted weights.

    ojas_rule(H,[I|Is],[{W,[]}|WPs],Output,Acc)->
            Updated_W = functions:saturation(W + H*Output*(I - Output*W),?SAT_LIMIT),
            ojas_rule(H,Is,WPs,Output,[{Updated_W,[H]}|Acc]);
    ojas_rule(_H,[],[],_Output,Acc)->
            lists:reverse(Acc).
%ojas_rule/5 updates every synaptic weight using the Oja's learning rule.
```

With the implementation of this learning rule complete, we now move forward and discuss neural plasticity through neuromodulation.

15.3 Neuromodulation

Thus far we have discussed and implemented the Hebbian learning, which is a homosynaptic plasticity (also known as homotropic modulation) method, where the synaptic strength changes based on its history of activation. It is a synaptic weight update rule which is a function of its post- and pre- synaptic activity, as shown in Fig-15.2. But research shows that there is another approach to synaptic plasticity which nature has discovered, a highly dynamic and effective one, plasticity through neuromodulation.

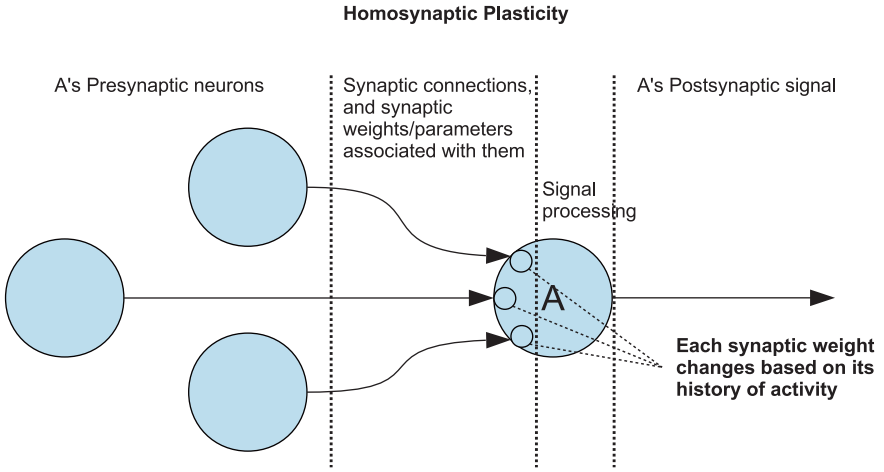


Fig. 15.2 Homosynaptic mechanism for Neuron A's synaptic weight updating, based on the pre- and post- synaptic activity of neuron A.

Neuromodulation is a form of heterosynaptic plasticity. In heterosynaptic plasticity the synaptic weights are changed due to the synaptic activity of other neurons, due to the modulating signals other neurons can produce to affect the given neuron's synaptic weights. For example, assume we have a neural circuit composed of two neurons, a presynaptic neuron N1, and a postsynaptic neuron N2. There can be other neurons N3, N4... which also connect to N2, but their neurotransmitters affect N2's plasticity, rather than being used as signals on which the N2's output signal is based on. The accumulated signals, neurotransmitters, from N3, N4..., could then dictate how rapidly and in what manner N2's connection strengths change. This type of architecture is shown in Fig-15.3.

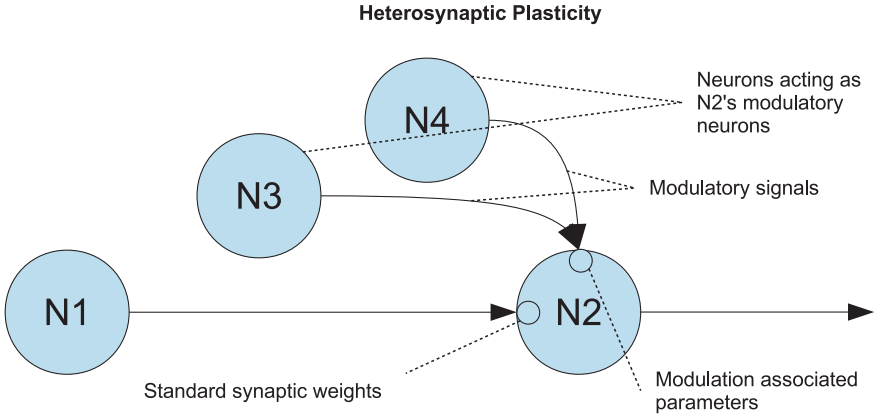
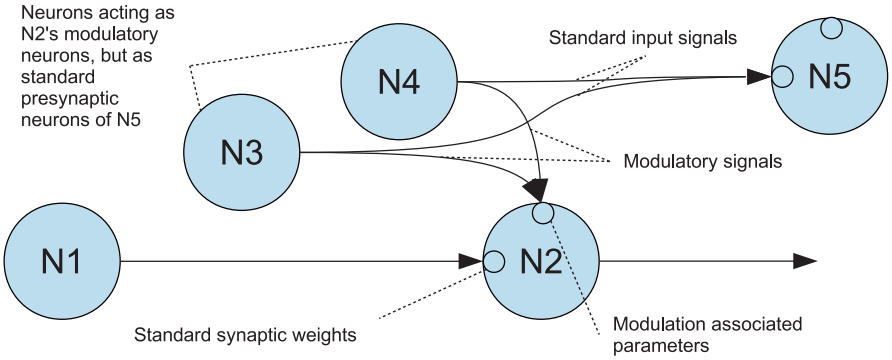


Fig. 15.3 Heterosynaptic mechanism for plasticity, where the Hebbian plasticity is modulated by a modulatory signal from neurons N3 and N4.

If we assume the use of the Generalized Hebbian learning rule for the synaptic weight update rule: Updated_W_i= W_i + h*(A*I_i*Output + B*I_i + C*Output + D), then the accumulated neuromodulatory signals from the other neurons could be used to calculate the learning parameter h, with the parameters A, B, C, and D evolved and specified within the postsynaptic neuron N2. In addition, the neuromodulatory signals from neurons N3, N4... could also be used to modulate and specify the parameters A, B, C, and D, as well.

The modulating neurons could be standard neurons, and whether their output signals are used as modulatory signals, or standard input signals, could be determined fully by the postsynaptic neuron to which they connect, as shown in Fig-15.4.



It should be noted that both, the modulatory and the standard output signals of the neurons N3 and N4, are the same neural output signals.

Fig. 15.4 Input signals used as standard signals, and as modulatory signals, dependent on how the postsynaptic neuron decides to treat the presynaptic signals.

Another possible approach is to set-up *secondary* neurons to the postsynaptic neuron N2 which we want modulated, where the secondary neurons receive exactly the same input signals as the postsynaptic neuron N2, but the output signals of these secondary neurons are used as modulatory signals of N2. This type of topological and architectural setup is shown in Fig-15.5.

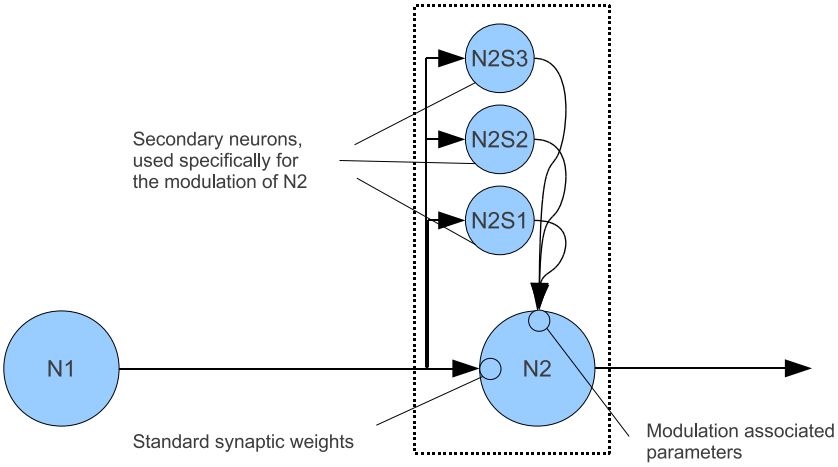


Fig. 15.5 Secondary neurons, created and used specifically for neuromodulation.

Through the use of dedicated modulatory neurons, it is possible to evolve whole modulatory networks. Complex systems whose main role is to modulate another neural network's plasticity and learning, its long-term potentiation, its ability to form memory. In this method, the generated learning parameter is signal specific, and itself changes; the learning ability and form evolves with everything else. Unlike the simple Hebbian or Oja's learning rule, these plasticity systems would depend on the actual input signals, on the sensory signals, and other regulatory and processing parts of the neural network system, which is a much more biologically faithful neural network architecture, and would allow our system to evolve even more complex behaviors.

Nature uses a combination of the architectures shown in figures 15.1 through 15.5. We have already discussed the Hebbian learning rule, and implemented the architecture of Fig-15.2. We now add the functionality to give our neuroevolutionary system the ability to evolve NN systems with architectures shown in Fig-15.4 and Fig-15.5. This will give our systems the ability to evolve self adaptation, and learning.

15.3.1 The Neuromodulatory Architecture

The architecture in Fig-15.5 could be easily developed using our already existing architecture, and it would even increase the ratio of neural computations performed by the neuron to the number of signals sent to the neuron. This is important because Erlang becomes more effective with big computations and small messages. The way we can represent this architecture is through the *weight_parameters* based approach. The weight_parameters could be thought of

as synaptic weights themselves, but for the secondary neurons. These secondary neurons share the process of the neuron they are to modulate, and because the secondary neurons need to process the same input vectors that the neuron they are modulating is processing, it makes this design highly efficient. This architectural implementation is shown in Fig-15.6.

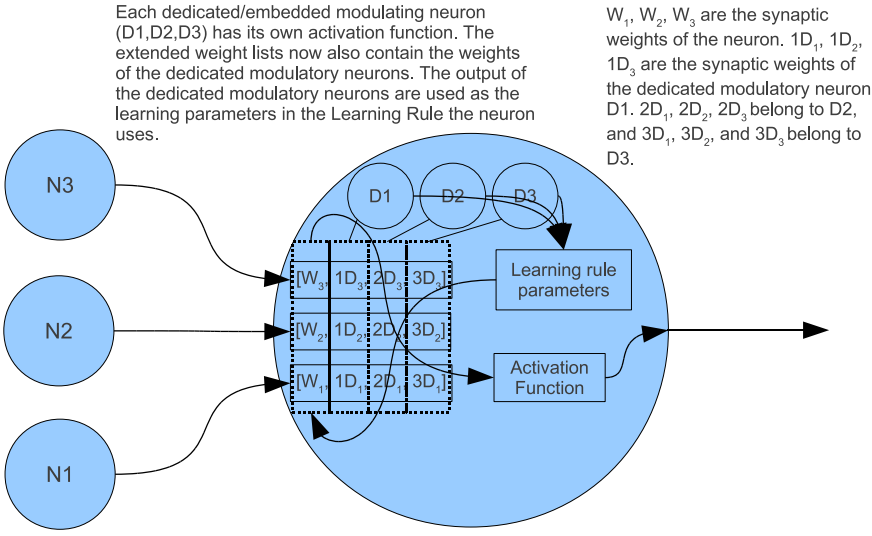


Fig. 15.6 The architectural implementation of neuromodulation through dedicated/embedded modulating neurons.

In the above figure we see three neurons: N1, N2, and N3, connected to another neuron, which is expanded in the figure and whose architecture is shown. This neuron has a standard activation function, and a learning rule, but its *input_idps* list is extended. What we called parameters in the other learning rules, are here used as synaptic weights belonging to this neuron's embedded/dedicated modulating neurons: D_1, D_2, and D_3. Furthermore, each dedicated/embedded modulating neuron (D_1,D_2,D_3) can have its own activation function, but usually just uses the *tanh* function.

If each weight parameter list is of length 1, then there is only a single dedicated modulating neuron, and the dedicated neuron's output can be designated as the learning parameter: *h*. The learning parameters *A*, *B*, *C*, and *D*, can be specified by the *neural_parameters* list. Or we can have the weight parameters list be of size 2, and thus specify 2 dedicated modulating neurons, whose outputs would dictate the learning parameters *h* and *A*, with the other parameters specified in the *neural_parameters* list. Finally, we can have the weight parameters list be of length 5, thus representing the synaptic weights of 5 dedicated modulating neurons, whose outputs specify all the parameters (h, A, B, C, D) of the General Hebbian learning rule.

Having 5 separate dedicated modulating neurons does have its problems though, because it magnifies the number of synaptic weights/parameters our neuroevolutionary system has to tune, mutate, and set up. If our original neuron, without plasticity, had a synaptic weight list of size 10, this new modulated neuron would have 60 synaptic weight parameters for the same 10 inputs. All of these parameters would somehow have to be specified, tuned, and made to work perfectly with each other, and this would all only be a single neuron. Nevertheless, it is an efficient implementation of the idea, and would be easy to add due to the way our neuroevolutionary system's architecture is set up.

To allow for general neuromodulation (Fig-15.3), so that the postsynaptic neuron can designate some of the presynaptic signals as holding standard information, and others as holding modulatory information, could be done in a number of ways. Let us consider two of such approaches next:

1. This approach would require us adding a new element to the neuron record, akin to input_idps. We could add a secondary such element and designate it *input_idps_modulation*. It too would be represented as a list of tuples: *[{Input_Id,Weight}...]*, but the resulting computed dot product, sent through its own activation function, would be used as a learning parameter. But which of the learning parameters? H, A, B, C, or D? The standard approach is to use the following equation: *Updated_W = M_Output*H*(A*I*Output + B*Output + C*Output + D)*, where M_Output is the output signal produced by processing the input signals using the synaptic weights specified in the *input_idps_modulation list*, and where the parameters H, A, B, C, and D are simply neural_parameters, and as other parameters can be perturbed and evolved during the tuning phase and/or during the topological mutation phase.

How would the post synaptic neuron decide whether the new connection (added during the topological mutation phase) should be used as a standard signal, and thus be added to the input_idps list, or as modulatory input signal, and thus added to input_idps_modulation list? We could set up a rule so that if the neuron is designated to have general modulation based plasticity, the very first connection to the neuron is designated as standard input, and then any new connections are randomly sorted into either the input_idps or input_idps_modulation lists. To add this approach would only require adding a new list, and we would already have all the necessary functions to mutate its parameters, to clone it during neuronal cloning process, and to process input signals, because this new list would be exactly like the input_idps list. The overhead of simply adding this extra parameter, input_idps_modulation, to the neuron record, would be minuscule, and this architecture is what was represented in Fig-15.4.

2. Another way a neuron could decide on whether the presynaptic signal sent to it is standard or modulatory, is by us having neuronal types, where some neurons are type: *standard*, and others are type: *modulatory*. The signals sent by modulatory neurons are *always* used by all postsynaptic neurons for modulating the generalized Hebbian plasticity rule. The architecture of this type of system is

shown in Fig-15.7. In this figure I show a NN topology composed of standard neurons (std), and modulatory neurons (mod). They are all interconnected, each can receive signals from any other. The difference in how those signals are processed is dependent on the presynaptic neuron's type. If it is of type mod, then it is used as modulatory, if it is type std, then it is used as a standard input signal. Modulatory neurons can even modulate other modulatory neurons, while the outputs of the standard neurons can be used by both standard and modulatory neurons.

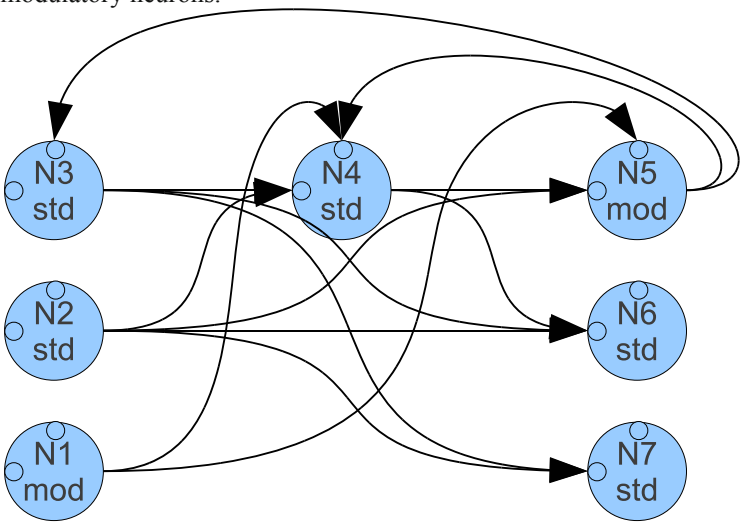


Fig. 15.7 A topology of a heterosynaptic, general, neural network system with neurons of type standard (std) and modulatory (mod).

3. But the first and second implementation does not solve the problem that the Hebbian learning rule uses multiple parameters, and we want to have the flexibility to specify 1 or more of them, based on the incoming modulatory signals. Another solution that does solve this is by tagging input signals with tags *i*, *h*, *a*, *b*, *c*, *d*, where *i* tags the standard inputs, and *h*, *a*, *b*, *c*, and *d*, tag the modulatory input signals associated with the tag named modulating learning parameter. Though this may at first glance seem like a more complex solution, we actually already have solved it, and it would require us only changing a few functions.

We are already generating weight based parameters. Thus far they have been lists, but they can also be atomic tags as follows: *[{Input_PId, [{Weight1,Tag1}, {Weight2,Tag2}...]}...]*. This is a clean solution that would allow us to designate different incoming signals to be used for different things. Mutation operators would not need to be modified significantly either, we would simply add a clause stating that if the neuron uses the *general_modulation* plasticity function, then the Tag is generated randomly from the following list: *[i, h, a, b, c, d]*. The most significant modification would have to be done to the signal_aggregation function,

since we would need to sort the incoming signals based on their tags, and then calculate the different output signals based on their tags, with the *i* output signal being the standard one produced by the postsynaptic neuron, and the *h*, *a*, *b*, *c*, and *d*, output signals being used as modulatory learning parameters. But even that could be isolated to just the plasticity function, which has access to the IAcc, Input_PIdPs, and everything else necessary to compute output signals. The architecture of a neuron using this approach to general neuromodulation is shown in Fig-15.8.

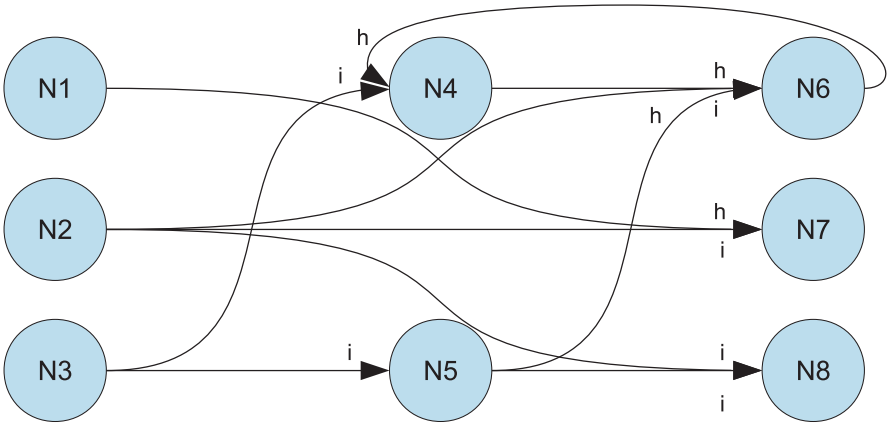


Fig. 15.8 Tag based architecture of a general neuromodulation capable neural network.

What is the computational difference between all of these neuromodulation approaches? How would the neural networks act differently when evolved with one approach rather than another? Would it even be possible to see the difference? Should we implement them all, provide all of these options to the neuroevolutionary system in hopes that it can sort things out on its own, and use the best one (throwing everything at the wall, and see what sticks)? How do we test which of these plasticity type architectures is better? How do we define "better"? Do we define it as the NN evolving faster (the neuroevolutionary system taking less number of evaluations to evolve a solution for some given problem)? Or do we define better as having the evolved NNs more dynamic, more adaptive, more general, but evolved slower due to so many different parameters for the evolutionary process to having to deal with? These are all open research questions.

We cannot test the effectiveness of plasticity enabled neural network systems on the standard double pole balancing, xor, or clustering type of benchmarks and tests. To test how well a plasticity enabled NN system functions, we need to apply our neuroevolutionary system to a problem where environment changes, where adaptation and learning over time gives an advantage. We could test plasticity by using it in the ALife simulation, T-Maze and double T-Maze navigation [2,3], or by applying it to some other robotics & complex navigation project. Though the small differences between these various modulatory approaches might require a lot of work to see, since evolution will tend to go around any small problems

posed by any one implementation or architecture over another. Nevertheless, the fact that it is so easy for us to implement, test, and research these advanced learning rules and plasticity approaches, means that we **can** find out, we can determine what works better, and what approach will yield a more general, more intelligent, neural network based agent. If our system were not have been written in Erlang, adding neuroplasticity would have posed a much greater problem.

We will implement the dedicated neuromodulators (where the weight parameters represent the synaptic weights of embedded secondary neurons, whose output dictates the parameters of the general Hebbian learning rule), and the general neuromodulation plasticity through the use of the input_idps_modulation element. Our plasticity function using the first of these two approaches will be called: *self_modulation*, and the second: *general_modulation.* In the next section we will further define and implement these neuromodulatory based learning rules.

15.3.2 Implementing the self_modulation Learning Rules

We will first implement the *self_modulation* plasticity function. Given the general Hebbian learning rule for synaptic weight updating: Updated_W_i = W_i + H*(A*I_i*Output + B*I_i + C*Output + D), we can have multiple versions of this function. Version-1: where the secondary embedded neuron only outputs the H learning parameter, with the parameter A set to some predetermined constant value within the neural_parameters list, and B=C=D=0. Version-2: where A is generated randomly when generating the neural_parameters list, and B=C=D=0. Version-3: where B, C, and D are also generated randomly in the *neural_parameters* list. Version-4: where the weight_parameters generates a list of length 2, thus allowing the neuron to have 2 embedded modulatory neurons, one outputting a parameter we use for H, and another outputting the value we can use as A, with B=C=D=0. Version-5: Where B, C, and D are generated randomly by the PlasticityFunctionName(neural_parameters) function. And finally Version-6: Where the *weight_parameters* produces a list of length 5, allowing the neuron to have 5 embedded modulatory neurons, whose outputs are used for H, A, B, C, and D. All of these variations will have most of their functionality shared, and thus will be quick and easy to implement.

The *self_modulationV1*, *self_modulationV2*, and *self_modulationV3* are all very similar, mainly differing in the parameter lists returned by the *PlasticityFunctionName(neural_parameters)* function, as shown in Listing 15.6. All three of these plasticity functions use the neuromodulation/5 function which accepts the H, A, B, C, and D learning parameters, and updates the synaptic weights of the neuron using the general Hebbian rule: $Updated\_W_i = W_i + H*(A*I_i*Output + B*I_i + C*Output + D)$.

Listing-15.6 The self_modulationV1-3 functions of arity 1, generating the neural and weight parameters.

```
self_modulationV1(neural_parameters)->
    A=0.1,
    B=0,
    C=0,
    D=0,
    [A,B,C,D];
self_modulationV1(weight_parameters)->
    [(lists:random()-0.5)].

self_modulationV1([A,B,C,D],IAcc,Input_PIdPs,Output)->
    H = math:tanh(dot_productV1(IAcc,Input_PIdPs)),
    neuromodulation([H,A,B,C,D],IAcc,Input_PIdPs,Output,[]).

    dot_productV1(IAcc,IPIdPs)->
            dot_productV1(IAcc,IPIdPs,0).
    dot_productV1([{IPId,Input}|IAcc],[{IPId,WeightsP}|IPIdPs],Acc)->
            Dot = dotV1(Input,WeightsP,0),
            dot_productV1(IAcc,IPIdPs,Dot+Acc);
    dot_productV1([],[{bias,[{_Bias,[H_Bias]}]}],Acc)->
            Acc + H_Bias;
    dot_productV1([],[],Acc)->
            Acc.

            dotV1([I|Input],[{_W,[H_W]}|Weights],Acc) ->
                    dotV1(Input,Weights,I*H_W+Acc);
            dotV1([],[],Acc)->
                    Acc.

neuromodulation([H,A,B,C,D],[{IPId,Is}|IAcc],[{IPId,WPs}|Input_PIdPs],Output,Acc)->
    Updated_WPs = genheb_rule([H,A,B,C,D],Is,WPs,Output,[]),
    neuromodulation([H,A,B,C,D],IAcc,Input_PIdPs,Output,[{IPId,Updated_WPs}|Acc]);
neuromodulation(_NeuralParameters,[],[],_Output,Acc)->
    lists:reverse(Acc);
neuromodulation([H,A,B,C,D],[],[{bias,WPs}],Output,Acc)->
    Updated_WPs = genheb_rule([H,A,B,C,D],[1],WPs,Output,[]),
    lists:reverse([{bias,Updated_WPs}|Acc]).

    genheb_rule([H,A,B,C,D],[I|Is],[{W,Ps}|WPs],Output,Acc)->
            Updated_W = functions:saturation(W + H*(A*I*Output + B*I + C*Output + D),
?SAT_LIMIT),
            genheb_rule(H,Is,WPs,Output,[{Updated_W,Ps}|Acc]);
    genheb_rule(_H,[],[],_Output,Acc)->
```

```
            lists:reverse(Acc).

self_modulationV2(neural_parameters)->
    A=(lists:random()-0.5),
    B=0,
    C=0,
    D=0,
    [A,B,C,D];
self_modulationV2(weight_parameters)->
    [(lists:random()-0.5)].

self_modulationV2([A,B,C,D],IAcc,Input_PIdPs,Output)->
    H = math:tanh(dot_productV1(IAcc,Input_PIdPs)),
    neuromodulation([H,A,B,C,D],IAcc,Input_PIdPs,Output,[]).

self_modulationV3(neural_parameters)->
    A=(lists:random()-0.5),
    B=(lists:random()-0.5),
    C=(lists:random()-0.5),
    D=(lists:random()-0.5),
    [A,B,C,D];
self_modulationV3(weight_parameters)->
    [(lists:random()-0.5)].

self_modulationV3([A,B,C,D],IAcc,Input_PIdPs,Output)->
    H = math:tanh(dot_productV1(IAcc,Input_PIdPs)),
    neuromodulation([H,A,B,C,D],IAcc,Input_PIdPs,Output,[]).
```

The self_modulationV4 – V5 differ only in that the weight_parameters is a list of length 2, and the A parameter is no longer specified in the neural_parameters list, and is instead calculated by the second dedicated modulatory neuron. The self_modulationV6 function on the other hand specifies the neural_Parameters as an empty list, and the weight_parameters list is of length 5, a single weight for every embedded modulatory neuron. The implementation of self_modulationV6 is shown in Listing-15.7.

Listing-15.7 The implementation of the self_modulationV6 plasticity function, composed of 5 embedded modulatory neurons.

```
self_modulationV6(neural_parameters)->
    [];
self_modulationV6(weight_parameters)->
    [(lists:random()-0.5),(lists:random()-0.5),(lists:random()-0.5), (lists:random()-0.5),
(lists:random()-0.5)].
```

```
self_modulationV6(_Neural_Parameters,IAcc,Input_PIdPs,Output)->
    {AccH,AccA,AccB,AccC,AccD} = dot_productV6(IAcc,Input_PIdPs),
    H = math:tanh(AccH),
    A = math:tanh(AccA),
    B = math:tanh(AccB),
    C = math:tanh(AccC),
    D = math:tanh(AccD),
    neuromodulation([H,A,B,C,D],IAcc,Input_PIdPs,Output,[]).

    dot_productV6(IAcc,IPIdPs)->
            dot_productV6(IAcc,IPIdPs,0,0,0,0,0).
    dot_productV6([{IPId,Input}|IAcc],[{IPId,WeightsP}|IPIdPs],AccH,AccA,AccB,AccC,
AccD)->
            {DotH,DotA,DotB,DotC,DotD} = dotV6(Input,WeightsP,0,0,0,0,0),
    dot_productV6(IAcc,IPIdPs,DotH+AccH,DotA+AccA,DotB+AccB,DotC+AccC,DotD
+AccD);
    dot_productV6([],[{bias,[{_Bias,[H_Bias,A_Bias,B_Bias,C_Bias,D_Bias]}]}],AccH,AccA,
AccB,AccC,AccD)->
            {AccH + H_Bias,AccA+A_Bias,AccB+B_Bias,AccC+C_Bias,AccD+D_Bias};
    dot_productV6([],[],AccH,AccA,AccB,AccC,AccD)->
            {AccH,AccA,AccB,AccC,AccD}.

    dotV6([I|Input],[{_W,[H_W,A_W,B_W,C_W,D_W]}|Weights],AccH,AccA,AccB,AccC,
AccD) ->
    dotV6(Input,Weights,I*H_W+AccH,I*A_W+AccA,I*B_W+AccB,I*C_W+AccC,I*D_W+
AccD);
    dotV6([],[],AccH,AccA,AccB,AccC,AccD)->
            {AccH,AccA,AccB,AccC,AccD}.
```

The architecture of the neuron using this particular plasticity function is shown in Fig-15.9. Since every synaptic weight of this neuron has a complementary parameter list of length 5, with an extra synaptic weight for every secondary, embedded modulatory neuron that analyzes the same signals as the actual neuron, but whose output signals modulate the plasticity of the neuron, each neuron thus has *x5* number of parameters (synaptic weights) that need to be tuned. This might be a price too high to pay by amplifying the curse of dimensionality. The more parameters that one needs to tune and set up concurrently, the more difficult it is to find a good combination of such parameters. Nevertheless, the generality it provides, and the ability to use a single process to represent multiple embedded modulatory neurons, has its benefits in computational efficiency. Plus, our system does after all try to alleviate the curse of dimensionality through *Targeted Tuning*, by concentrating on the newly added and affected neurons of the NN system. And thus we might just be on the edge of this one.

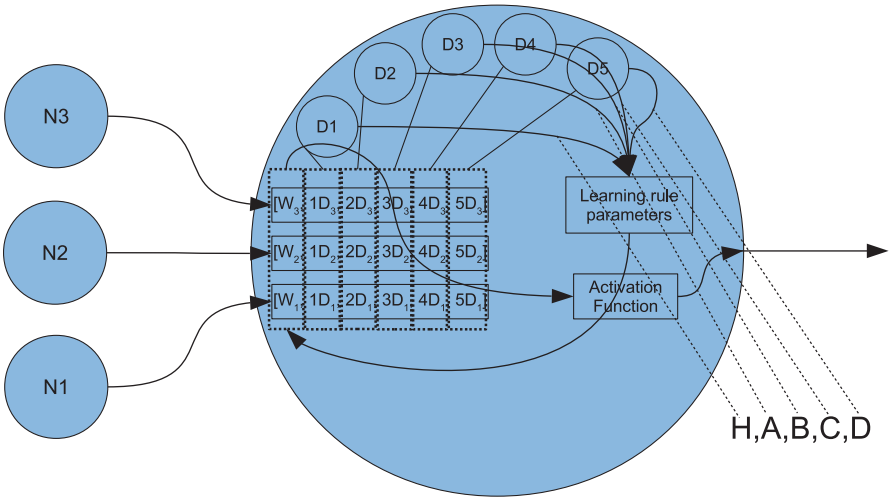


Fig. 15.9 The architecture of the neuron using self_modulationV6 plasticity function.

We noted earlier that there is another approach to neuromodulation, one that is more biologically faithful, in which a postsynaptic neuron uses some of the signals coming from the presynaptic neurons as modulatory signals, and others as standard signals. In the next section we will see what needs to be done to implement such a learning rule.

15.3.3 Implementing the input_idps_modulation Based Neuromodulated Plasticity

To implement neuromodulation using this method, we first modify the neuron's record by adding the *input_idps_modulation* element to it. The input_idps_modulation element will have the same purpose and formating as the *input_idps* element, to hold a list of tuples of the form: {Input_PId, WeightP}. The Input_PIds will be associated with the elements that send the postsynaptic neuron its modulatory signals, with the WeightP being of the same format as in the input_Idps list.

This particular implementation of neuromodulation will not require a lot of work, due to the input_idps_modulation list having a format which we already can process with the developed functions. The neuron cloning function in the genotype can be used to clone this list, the Id to PId conversion performed by the exoself to compose the Input_PIdPs list is also viable here. Even the synaptic weight perturbation can be applied to this list, due to it having such a similar format. The main changes we have to perform are to the neuron's main loop.

We must convert the neuron's main loop such that it can support 2 Input_PId lists, the SI_PIds (standard input PId list), and the MI_PIds (modulatory input PId

list), in the same way that the original neuron implementation supported the single Input_PIds list created from the Input_PIdPs. With these two lists we can then aggregate the input signals, and sort them either in to the standard input signal accumulator, or the modulatory signal accumulator, dependent on whether the incoming signal was coming from an element with an SI_PId or an MI_PId.

To make the implementation and the source code cleaner, we will create a state record for the neuron, which will contain all the necessary elements it requires for operation:

```
-record(state,{
    id,
    cx_pid,
    af,
    pf,
    aggrf,
    si_pids=[],
    si_pidps_current=[],
    si_pidps_backup=[],
    mi_pids=[],
    mi_pidps_current=[],
    mi_pidps_backup=[],
    output_pids=[],
    ro_pids=[]
}).
```

With this state record, we update the prep/1 function to use it, and clean the original loop function to hide all the non-immediately used lists and data in the state record. As in the original neuron process implementation, we have to create the Input_PId list so that the incoming signals can be sorted in the same order that the Input_PIdPs are sorted. This time though, we have two such lists, designated as the SI_PIdPs (the standard one), and the MI_PIdPs (the modulatory one). Thus we create two PId lists for the loop.

The main problem here is that as the neuron accumulates its input signals, one of these PId lists will empty out first, which would require a new clause to deal with it, since our main loop uses: [SI_PId|SI_PIds],[MI_PId|MI_PIds]. We did not have such a problem when we only used a single list, because when that list emptied out, the signal accumulation was finished. To avoid having to create a new clause, we add the atom *ok* to the end of both PId lists, and put the clause: *loop(S,ExoSelf_PId,[ok],[ok],SIAcc,MIAcc)* above the main loop. Because of the *ok* atom at the end of both lists, neither goes empty, letting us keep a single clause with the final state for both lists being *[ok],* which is achieved after the neuron has accumulated all the incoming standard and modulatory signals. The only problem with this setup is that the first clause is always pattern matched before the main loop, making the neuron process slower and less efficient. There are other ways to

implement this, and we could even set up two different main process loops, one for when the neuron uses neuromodulation, and one for when it does not (and thus needing only a single PId list). But this implementation is the most concise, and cleanest. The neuron process can always be optimized later on. The modified prep/1 function, and the neuron's new main loop, are shown in Listing-15.8.

Listing-15.8 The updated implementation of the neuron process.

```
prep(ExoSelf_PId) ->
    random:seed(now()),
    receive
        {ExoSelf_PId,{Id,Cx_PId,AF,PF,AggrF,SI_PIdPs,MI_PIdPs,Output_PIds,
RO_PIds}} ->
            fanout(RO_PIds,{self(),forward,[?RO_SIGNAL]}),
            SI_PIds = lists:append([IPId || {IPId,_W} <- SI_PIdPs, IPId =/= bias],[ok]),
            MI_PIds = lists:append([IPId || {IPId,_W} <- MI_PIdPs, IPId =/= bias],[ok]),
            io:format("SI_PIdPs:~p ~nMI_PIdPs:~p~n",[SI_PIdPs,MI_PIdPs]),
            S=#state{
                id=Id,
                cx_pid=Cx_PId,
                af=AF,
                pf=PF,
                aggrf=AggrF,
                si_pids=SI_PIds,
                si_pidps_current=SI_PIdPs,
                si_pidps_backup=SI_PIdPs,
                mi_pids=MI_PIds,
                mi_pidps_current=MI_PIdPs,
                mi_pidps_backup=MI_PIdPs,
                output_pids=Output_PIds,
                ro_pids=RO_PIds
            },
            loop(S,ExoSelf_PId,SI_PIds,MI_PIds,[],[])
    end.
%When gen/1 is executed, it spawns the neuron element and immediately begins to wait for its initial state message from the exoself. Once the state message arrives, the neuron sends out the default forward signals to any elements in its ro_ids list, if any. Afterwards, the prep function drops into the neuron's main loop.

loop(S,ExoSelf_PId,[ok],[ok],SIAcc,MIAcc)->
    PF = S#state.pf,
    AF = S#state.af,
    AggrF = S#state.aggrf,
    {PFName,PFParameters} = PF,
    Ordered_SIAcc = lists:reverse(SIAcc),
```

```
    SI_PIdPs = S#state.si_pidps_current,
    SAggregation_Product = signal_aggregator:AggrF(Ordered_SIAcc,SI_PIdPs),
    SOutput = functions:AF(SAggregation_Product),
    Output_PIds = S#state.output_pids,
    [Output_PId ! {self(),forward,[SOutput]} || Output_PId <- Output_PIds],

    Ordered_MIAcc = lists:reverse(MIAcc),
    MI_PIdPs = S#state.mi_pidps_current,
    MAggregation_Product = signal_aggregator:dot_product(Ordered_MIAcc,MI_PIdPs),
    MOutput = functions:tanh(MAggregation_Product),
    U_SI_PIdPs = plasticity:PFName([MOutput|PFParameters],Ordered_SIAcc,SI_PIdPs,
SOutput),
    U_S=S#state{
            si_pidps_current = U_SI_PIdPs
    },
    SI_PIds = S#state.si_pids,
    MI_PIds = S#state.mi_pids,
    loop(U_S,ExoSelf_PId,SI_PIds,MI_PIds,[],[]);
loop(S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc)->
    receive
            {SI_PId,forward,Input}->
                    loop(S,ExoSelf_PId,SI_PIds,[MI_PId|MI_PIds],[{SI_PId,Input}|SIAcc],
MIAcc);
            {MI_PId,forward,Input}->
                    loop(S,ExoSelf_PId,[SI_PId|SI_PIds],MI_PIds,SIAcc,[{MI_PId,Input}|
MIAcc]);
            {ExoSelf_PId,weight_backup}->
                    U_S = S#state{
                            si_pidps_backup=S#state.si_pidps_current,
                            mi_pidps_backup=S#state.mi_pidps_current
                    },
                    loop(U_S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc);
            {ExoSelf_PId,weight_restore}->
                    U_S = S#state{
                            si_pidps_current=S#state.si_pidps_backup,
                            mi_pidps_current=S#state.mi_pidps_backup
                    },
                    loop(U_S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc);
            {ExoSelf_PId,weight_perturb,Spread}->
                    Perturbed_SIPIdPs=perturb_IPIdPs(Spread,S#state.si_pidps_backup),
                    Perturbed_MIPIdPs=perturb_IPIdPs(Spread,S#state.mi_pidps_backup),
                    U_S = S#state{
                            si_pidps_current=Perturbed_SIPIdPs,
                            mi_pidps_current=Perturbed_MIPIdPs
                    },
```

```
                loop(U_S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc);
        {ExoSelf_PId,reset_prep}->
                neuron:flush_buffer(),
                ExoSelf_PId ! {self(),ready},
                RO_PIds = S#state.ro_pids,
                receive
                        {ExoSelf_PId, reset}->
                                fanout(RO_PIds,{self(),forward,[?RO_SIGNAL]})
                end,
                loop(S,ExoSelf_PId,S#state.si_pids,S#state.mi_pids,[],[]);
        {ExoSelf_PId,get_backup}->
                NId = S#state.id,
                ExoSelf_PId ! {self(),NId,S#state.si_pidps_backup,S#state.mi_pidps_backup},
                loop(S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc);
        {ExoSelf_PId,terminate}->
                io:format("Neuron:~p is terminating.~n",[self()])
end.
```

With the implementation of the updated neuron now complete, we need to create the neuromodulation function in the plasticity module. Since the modulatory signals will be used to compute a nonlinear value used to modulate the standard general Hebbian rule, we will not need any weight_parameters and so our plasticity function will produce an empty weight_parameters list. But we will need the general neural_parameters for the *hebbian* function, thus the neuromodulation/1 function executed with the neuronal_parameters atom will return a list with 5 randomly generated (and later tuned and evolved) parameters: *[H,A,B,C,D]*. The neuromodulation/4 function is very simple, since it is executed with a list of all the necessary parameters to call the neurmodulation/5 function that applies the general hebbian rule to all the synaptic weights. These two added functions are shown in Listing-15.9.

Listing-15.9 The implementation of the neuromodulation/1 and neuromodulation/4 functions.

```
neuromodulation(neural_parameters)->
    H = (lists:random()-0.5),
    A = (lists:random()-0.5),
    B = (lists:random()-0.5),
    C = (lists:random()-0.5),
    D = (lists:random()-0.5),
    [H,A,B,C,D];
neuromodulation(weight_parameters)->
    [].

neuromodulation([M,H,A,B,C,D],IAcc,Input_PIdPs,Output)->
    Modulator = scale_dzone(M,0.33,?SAT_LIMIT),
    neuromodulation([Modulator*H,A,B,C,D],IAcc,Input_PIdPs,Output,[]).
```

The value M is the one computed by using the synaptic weights of the input_idps_modulation, using the dot_product signal aggregator, and the hyperbolic tangent (tanh) activation function. Since H scales the plasticity in general, multiplying the *Modulator* value by H allows for the modulation signal to truly modulate synaptic plasticity based on the parameters evolved by the neuron.

The *Modulator* value is computed by executing the scale_dzone/3 function, which performs 2 tasks:

1. Zero out M if it is between -0.33 and 0.33.
2. If M is greater than 0.33 or less than -0.33, normalize and scale it to be between 0 and ?SAT_LIMIT, or 0 and -?SAT_LIMIT, respectively.

This means that M has to reach a particular magnitude for the Hebbian rule to be executed, since when the Modulator value is 0 and is multiplied by H, the weights are not updated. The scale_dzone/3 function, and its supporting function, are shown in Listing-15.10.

Listing-15.10 The implementation of scale_dzone and scale function.

```
scale_dzone(Val,Threshold,MaxMagnitude)->
    if
            Val > Threshold ->
                    (functions:scale(Val,MaxMagnitude,Threshold)+1)*MaxMagnitude/2;
            Val < -Threshold ->
                    (functions:scale(Val,-Threshold,-MaxMagnitude)-1)*MaxMagnitude/2;
            true ->
                    0
    end.

    scale(Val,Max,Min)->
            case Max == Min of
                    true ->
                            0;
                    false ->
                            (Val*2 - (Max+Min))/(Max-Min)
            end.
%The scale/3 function scales Val to be between -1 and 1, with the scaling dependent on the
Max and Min value, using the equation: Scaled_Val = (Val*2 - (Max + Min))/(Max-Min). The
function scale_dzone/3 zeroes the Val parameter if it is below the threshold, and scales it to be
between Threshold and MaxMagnitude if it is above the threshold.
```

Though we have now successfully implemented the autoassociative learning rules, and neuromodulation, we cannot use those features until we create the necessary tuning and mutation operators, such that our neuroevolutionary system can actually tune in the various learning parameters, and add the synaptic weights

needed by the neuromodulation functionality. We discuss and implement these necessary features in the next section.

15.4 Plasticity Parameter Mutation Operators

For the plasticity based learning rules to be useful, our neuroevolutionary system must be able to optimize them. For this we need to create new mutation operators. Though we could add the new mutation operators to the genome_mutator module, we will do something different instead. Since each plasticity function has its own restrictions (which learning parameters can/should be modified, and which can/should not be), and because there are so many of the different variants, and many more to be added as time goes on, it would not be effective to create these mutation operators inside the genome_mutator module. The genome_mutator should concentrate on the standard topology oriented mutation operators.

To more effectively handle this, we can offload these specialized mutation operators in the same way we offloaded the generation of the initial plasticity parameters, to the plasticity module itself. We can add a single mutation operator *mutate_plasticity,* which when executed, executes the *plasticity:PFName(Agent_Id, mutate)* function. Then the researcher which created the various plasticity function variants and types, can also create the mutation operator functions for it, whether they simply perturb neural level learning parameters, synaptic weight level parameters, or perform a more complex mutation. And of course if the plasticity function is set to none, we will have the function *plasticity:none(Agent_Id,mutate)* execute: *exit("Neuron does not support plasticity.")*, which will allow our neuroevolutionary system to attempt another mutation operator, without wasting the topological mutation try.

The plasticity specializing mutation operators should perform the following general operations:

- If the neuron uses neural_parameters, randomly choose between 1 and math:sqrt(TotParameters) number of parameters, and perturb them with a value selected randomly between -Pi and Pi.
- If the neuron uses weight_parameters, randomly choose between 1 and math:sqrt(TotWeightParameters) number of parameters, and perturb them with a value selected randomly between -Pi and Pi.
- If the neuron uses both, neural_parameters and weight_parameters, randomly choose one or the other, and perturb that parameter list using one of the above approaches, depending which of the two apply.

The neuromodulation is a special case, since it does not only have the global neural_level parameters which can be mutated/perturbed using the standard method listed above, but also allows for the establishment of new modulatory connections. Because the input_idps_modulation list has the same format as the standard

input_idps list, we can use the already existing synaptic connection establishing mutation operators and functions. The only modification we need to make so that some of the connections are standard, and others are modulatory, is set a case such that if the neuron to which the connection is being established has neuromodulation enabled, then the choice of whether the new connection will be standard or modulatory is 50/50, and if there is no neuromodulation enabled, then only the standard connection is allowed.

15.4.1 Implementing the Weight Parameter Mutation Operator

We first create the mutation operators which are applied to the weight_parameters. This mutation operator, executed when the plasticity function is run with the parameter: *{N_Id,mutate}*, performs similarly to the standard *perturb_IPIdPs/2* function, but instead of mutating the synaptic weights, it operates on, and mutates the, parameter values. The probability for any weight parameter to be perturbed is *1/math:sqrt(TotParameters)*. The plasticity functions that only use weight_parameters are the **hebbian_w** and **ojas_w**. Because in both of these plasticity functions the same implementation for the mutator is used, only the hebbian_w/1 version is shown (the difference for the ojas_w version is that instead of hebbian_w({N_Id,mutate}), we have ojas_w({N_Id,mutate})). This implementation is shown in Listing-15.11.

Listing-15.11 Implementation of the plasticity function based weight_parameter mutation operators.

```
hebbian_w({N_Id,mutate})->
    random:seed(now()),
    N = genotype:read({neuron,N_Id}),
    InputIdPs = N#neuron.input_idps,
    U_InputIdPs=perturb_parameters(InputIdPs,?SAT_LIMIT),
    N#neuron{input_idps = U_InputIdPs};
hebbian_w(neural_parameters)->
    [];
hebbian_w(weight_parameters)->
    [(lists:random()-0.5)].
%hebbian_w/1 function produces the necessary parameter list for the hebbian_w learning rule to operate. The parameter list for the simple hebbian_w learning rule is a parameter list composed of a single parameter H: [H], for every synaptic weight of the neuron. When hebbian_w/1 is called with the parameter neural_parameters, it returns []. When hebbian_w/1 is executed with the {N_Id,mutate} tuple, the function goes through every parameter in the neuron's input_idps, and perturbs the parameter value using the specified spread (?SAT_LIMIT).

    perturb_parameters(InputIdPs,Spread)->
```

```
        TotParameters = lists:sum([lists:sum([length(Ps) || {_W,Ps} <- WPs]) || {_Input_Id,
WPs} <- InputIdPs]),
        MutationProb = 1/math:sqrt(TotParameters),
        [{Input_Id,[{W,perturb(Ps,MutationProb,Spread,[])}|| {W,Ps} <- WPs]} || {Input_Id,
WPs} <- InputIdPs].
%The perturb_parameters/2 function goes through every tuple in the InputIdPs list, extracts the
WeightPlus blocks for each input connection, calculates the total number of weight parameters
the neuron has, and from it the probability with which those parameters will be perturbed.
The function then executes perturb/4 to perturb the said parameters.

        perturb([Val|Vals],MutationProb,Spread,Acc)->
                case random:uniform() < MutationProb of
                        true ->
                                U_Val = sat((random:uniform()-0.5)*2*Spread+Val,Spread,
Spread),
                                perturb(Vals,MutationProb,Spread,[U_Val|Acc]);
                        false ->
                                perturb(Vals,MutationProb,Spread,[Val|Acc])
                end;
        perturb([],_MutationProb,_Spread,Acc)->
                lists:reverse(Acc).
%The perturb/5 function is executed with a list of values and a probability with which each
value has the chance of being perturbed. The function then goes through every value and per-
turbs it with the given probability.
```

15.4.2 Implementing the Neural Parameter Mutation Operator

We next create the mutation operators which are applied to the neural_parameters, which are lists of values. To accomplish this, we just make that list pass through a function which with some probability, *1/sqrt(ListLength)*, perturbs the values within it. We add such mutation operators to the plasticity functions which only use the neural_parameters. The following plasticity functions only use the neural_parameters: **hebbian**, **ojas**, and the **neuromodulation**. Since all 3 would use exactly the same implementation, only the neuromodulation/1 implementation is shown in Listing-15.12.

Listing-15.12 Implementation of the neural_parameters mutation operator.

```
neuromodulation({N_Id,mutate})->
    random:seed(now()),
    N = genotype:read({neuron,N_Id}),
    {PFName,ParameterList} = N#neuron.pf,
    MSpread = ?SAT_LIMIT*10,
```

```
    MutationProb = 1/math:sqrt(length(ParameterList)),
    U_ParameterList = perturb(ParameterList,MutationProb,MSpread,[]),
    U_PF = {PFName,U_ParameterList},
    N#neuron{pf=U_PF};
neuromodulation(neural_parameters)->
    H = (lists:random()-0.5),
    A = (lists:random()-0.5),
    B = (lists:random()-0.5),
    C = (lists:random()-0.5),
    D = (lists:random()-0.5),
    [H,A,B,C,D];
neuromodulation(weight_parameters)->
    [].
%neuromodulation/1 function produces the necessary parameter list for the neuromodulation
learning rule to operate. The parameter list for this learning rule is a list composed of parame-
ters H,A,B,C,D: [H,A,B,C,D]. When the function is executed with the {NId,mutate} parameter,
it calculates the perturbation probability of every parameter through the equation:
1/math:sqrt(length(ParameterList)), and then executes the perturb/5 function to perturb the ac-
tual parameters.
```

The above shown mutation operator, called by executing neuromodulation/1 with the parameter {N_Id,mutate}, uses the perturb/4 function from the weight_parameters based mutation operator which was shown in the previous listing, Listing-15.11.

15.4.3 Implementing the Hybrid, Weight & Neural Parameters Mutation Operator

Finally, we also have plasticity functions which have both, neural_parameters and weight_parameters. This is the case for example for the self_modulationV5, V3, and V2 learning rules. For these type of plasticity functions, we create a combination of the neural_parameters and weight_parameters mutation operators, as shown in Listing-15.13.

Listing-15.13 A hybrid of the neural_parameters and weight_parameters mutation operator, implemented here for the self_modulationV5 plasticity function.

```
self_modulationV5({N_Id,mutate})->
    random:seed(now()),
    N = genotype:read({neuron,N_Id}),
    {PFName,ParameterList} = N#neuron.pf,
    MSpread = ?SAT_LIMIT*10,
```

```
    MutationProb = 1/math:sqrt(length(ParameterList)),
    U_ParameterList = perturb(ParameterList,MutationProb,MSpread,[]),
    U_PF = {PFName,U_ParameterList},
    InputIdPs = N#neuron.input_idps,
    U_InputIdPs=perturb_parameters(InputIdPs,?SAT_LIMIT),
    N#neuron{pf=U_PF,input_idps=U_InputIdPs};
self_modulationV5(neural_parameters)->
    B=(lists:random()-0.5),
    C=(lists:random()-0.5),
    D=(lists:random()-0.5),
    [B,C,D];
self_modulationV5(weight_parameters)->
    [(lists:random()-0.5),(lists:random()-0.5)].
```

For this plasticity module, this is all that is needed, there are only these 3 variants. We now modify the genome_mutator module to include the *mutate_plasticity_parameters* mutation operator, and modify the functions which deal with linking neurons together, so that we can add the *modulatory* connection establishment functionality.

15.4.4 Updating the genome_mutator Module

Since our neuroevolutionary system can only apply to a population the mutation operators available in its constraint record, we first add the *{mutate_plasticity_parameters,1}* tag to the constraint's *mutation_operators* list. This means that the mutate_plasticity_parameter mutation operator has the same chance of being executed as any other mutation operator within the mutation_operators list. After having modified the constraint record, we add the *mutate_plasticity_parameters/1* function to the *genome_mutator* module. It is a simple mutation operator that chooses a random neuron from the NN, and through the execution of *plasticity:PFName({N_Id,mutate})* function, mutates the plasticity parameters of that neuron, if that neuron has plasticity. If the neuron does not have plasticity enabled, then the plasticity:none/1 function is executed, which exits the mutation operator, letting our neuroevolutionary system try another mutation. The implemented mutate_plasticity_parameters/1 function is shown in Listing-15.14.

Listing-15.14 The implementation of the mutate_plasticity_parameters mutation operator.

```
mutate_plasticity_parameters(Agent_Id)->
    A = genotype:read({agent,Agent_Id}),
    Cx_Id = A#agent.cx_id,
    Cx = genotype:read({cortex,Cx_Id}),
    N_Ids = Cx#cortex.neuron_ids,
```

```
    N_Id = lists:nth(random:uniform(length(N_Ids)),N_Ids),
    N = genotype:read({neuron,N_Id}),
    {PFName,_Parameters} = N#neuron.pf,
    U_N = plasticity:PFName({N_Id,mutate}),
    EvoHist = A#agent.evo_hist,
    U_EvoHist = [{mutate_plasticity_parameters,N_Id}|EvoHist],
    U_A = A#agent{evo_hist=U_EvoHist},
    genotype:write(U_N),
    genotype:write(U_A).
%The mutate_plasticity_parameters/1 chooses a random neuron from the NN, and mutates the
parameters of its plasticity function, if present.
```

Having implemented the mutation operator, we now look for the connection/synaptic-link establishing functions. We need to modify these functions because we want to ensure that if the neuron uses the neuromodulation plasticity function, then some of the new connections that are added to it through evolution, are randomly chosen to be modulatory connections rather than standard ones.

The functions that need to be updated are the following four:

- add_bias/1: Because the input_idps_modulation can also use a bias weight.
- remove_bias/1: Because the input_idps_modulation should also be able to rid itself of its bias.
- link_ToNeuron/4: Which is the function that actually establishes new links, and adds the necessary tuples to the input_idps list. We should be able to randomly choose whether to add the new tuple to the standard input_idps list, or the modulatory input_idps_modulation list.
- cutlink_ToNeuron/3: Which is the function which cuts the links to the neuron, and removes the synaptic weight containing tuple from the input_idps list. We should be able to randomly choose whether to remove such a tuple from the input_idps or input_idps_modulation list.

Again, because of the way we developed, and modularized the code in the genome_mutator module, almost everything with regards to linking is contained in the link_ToNeuron and cutlink_ToNeuron, so by just modifying those, and the add_bias/remove_bias functions, we will be done with the update.

Originally the add_bias/1 function checks whether the input_idps list already has a bias, and then adds a bias if it does not, and exits if it does. We now have to check whether input_idps and input_idps_modulation lists already have biases. To do this, we randomly generate a value by executing random:uniform(2), which generates either 1 or 2. If value 2 is generated, and the input_idps_modulation does not have a bias, we add one to it. Otherwise, if the input_idps list does not have a bias, we add one to it, and thus in the absence of neuromodulation based plasticity, probability of adding the bias to input_idps does not change. The modified add_bias mutation operator is shown in Listing-15.15.

Listing-15.15 The updated add_bias mutation operator.

```
add_bias(Agent_Id)->
    A = genotype:read({agent,Agent_Id}),
    Cx_Id = A#agent.cx_id,
    Cx = genotype:read({cortex,Cx_Id}),
    N_Ids = Cx#cortex.neuron_ids,
    N_Id = lists:nth(random:uniform(length(N_Ids)),N_Ids),
    Generation = A#agent.generation,
    N = genotype:read({neuron,N_Id}),
    SI_IdPs = N#neuron.input_idps,
    MI_IdPs = N#neuron.input_idps_modulation,
    {PFName,_NLParameters} = N#neuron.pf,
    case {lists:keymember(bias,1,SI_IdPs), lists:keymember(bias,1,MI_IdPs), PFName ==
neuromodulation, random:uniform(2)} of
            {_,true,true,2} ->
                    exit("********ERROR:add_bias:: This Neuron already has a modulatory bias
part.");
            {_,false,true,2} ->
                    U_MI_IdPs = lists:append(MI_IdPs,[{bias,[{random:uniform()-0.5,
plasticity:PFName(weight_parameters)}]}]),
                    U_N = N#neuron{
                            input_idps_modulation = U_MI_IdPs,
                            generation = Generation},
                    EvoHist = A#agent.evo_hist,
                    U_EvoHist = [{{add_bias,m},N_Id}|EvoHist],
                    U_A = A#agent{evo_hist=U_EvoHist},
                    genotype:write(U_N),
                    genotype:write(U_A);
            {true,_,_,1} ->
                    exit("********ERROR:add_bias:: This Neuron already has a bias in in-
put_idps.");
            {false,_,_,_} ->
                    U_SI_IdPs = lists:append(SI_IdPs,[{bias,[{random:uniform()-0.5,
plasticity:PFName(weight_parameters)}]}]),
                    U_N = N#neuron{
                            input_idps = U_SI_IdPs,
                            generation = Generation},
                    EvoHist = A#agent.evo_hist,
                    U_EvoHist = [{{add_bias,s},N_Id}|EvoHist],
                    U_A = A#agent{evo_hist=U_EvoHist},
                    genotype:write(U_N),
                    genotype:write(U_A)
    end.
```

The remove_bias is modified in the same manner, and only a few elements of the source code are changed. Like the add_bias, we update the link_ToNeuron/4 function to randomly choose whether to make the new link modulatory or standard, and only if the chosen list (either input_idps or input_idps_modulation), does not already have a link from the specified presynaptic element. The updated function is shown in Listing-15.16.

Listing-15.16 The updated link_ToNeuron/4 function.

```
link_ToNeuron(FromId,FromOVL,ToN,Generation)->
    ToSI_IdPs = ToN#neuron.input_idps,
    ToMI_IdPs = ToN#neuron.input_idps_modulation,
    {PFName,_NLParameters}=ToN#neuron.pf,
    case {lists:keymember(FromId,1,ToSI_IdPs),lists:keymember(FromId,1,ToMI_IdPs)} of
            {false,false} ->
                    case {PFName == neuromodulation, random:uniform(2)} of
                            {true,2} ->
                                    U_ToMI_IdPs = [{FromId,
genotype:create_NeuralWeightsP(PFName,FromOVL,[])}|ToMI_IdPs],
                                    ToN#neuron{
                                            input_idps = U_ToMI_IdPs,
                                            generation = Generation
                                    };
                            _ ->
                                    U_ToSI_IdPs = [{FromId,
genotype:create_NeuralWeightsP(PFName,FromOVL,[])}|ToSI_IdPs],
                                    ToN#neuron{
                                            input_idps = U_ToSI_IdPs,
                                            generation = Generation
                                    }
                    end;
            _ ->
                    exit("ERROR:add_NeuronI::[cannot add I_Id]: ~p already connected to ~p~n",
[FromId,ToN#neuron.id])
    end.
%link_ToNeuron/4 updates the record of ToN, so that it's updated to receive a connection from
the element FromId. The link emanates from element with the id FromId, whose output vector
length is FromOVL, and the connection is made to the neuron ToN. In this function, either the
ToN's input_idps_modulation or input_idps list is updated with the tuple {FromId, [{W_1,
WPs} ...{W_FromOVL,WPs}]}. Whether input_idps or input_idps_modulation is updated, is
chosen randomly. Then the neuron's generation is updated to Generation (the current, most re-
cent generation). After this, the updated ToN's record is returned to the caller. On the other
hand, if the FromId is already part of the ToN's input_idps or input_idps_modulation list (de-
pendent on which was randomly chosen), which means that the standard or modulatory link al-
ready exists between the neuron ToN and element FromId, this function exits with an error.
```

Finally, we update the cutlink_ToNeuron/3 function. In this case, since there can only be one link between two elements, we simply first check if the specified input link is specified in the input_idps, and cut it if it does. If it does not, we check the input_idps_modulation next, and cut it if this link is modulatory. If such a link does not exist in either of the two lists, we exit the mutation operator with an error, printing to console that the specified link does not exist, neither in the synaptic weights list, nor in the synaptic parameters list. The implementation of the cutlink_ToNeuron/3, is shown in Listing-15.17.

Listing-15.17 The cutlink_ToNeuron/3 implementation.

```
cutlink_ToNeuron(FromId,ToN,Generation)->
        ToSI_IdPs = ToN#neuron.input_idps,
        ToMI_IdPs = ToN#neuron.input_idps_modulation,
        Guard1 = lists:keymember(FromId, 1, ToSI_IdPs),
        Guard2 = lists:keymember(FromId, 1, ToMI_IdPs),
        if
                Guard1->
                        U_ToSI_IdPs = lists:keydelete(FromId,1,ToSI_IdPs),
                        ToN#neuron{
                                input_idps = U_ToSI_IdPs,
                                generation = Generation};
                Guard2 ->
                        U_ToMI_IdPs = lists:keydelete(FromId,1,ToMI_IdPs),
                        ToN#neuron{
                                input_idps = U_ToMI_IdPs,
                                generation = Generation};
                true ->
                        exit("ERROR[can not remove I_Id]: ~p not a member of
~p~n",[FromId,ToN#neuron.id])
        end.
%cutlink_ToNeuron/3 cuts the connection on the ToNeuron (ToN) side. The function first
checks if the FromId is a member of the ToN's input_idps list, if it's not, then the function
checks if it is a member of the input_idps_modulation list. If it is not a member of either, the
function exits with error. If FromId is a member of one of these lists, then that tuple is removed
from that list, and the updated ToN record is returned to the caller.
```

With these updates completed, the genome_mutator module is up to date. In the case that a plasticity is enabled in any neuron, the topological mutation phase will be able to mutate the plasticity function learning parameters, and add modulatory connections in the case the plasticity function is *neuromodulation*. The only remaining update we have to make is one to the tuning phase related functions.

15.5 Tuning of a NN which has Plastic Neurons

It can be argued whether both standard synaptic weights and modulatory synaptic weights should be perturbed at the same time when the neuron has plasticity enabled, or just one or the other separately during the tuning phase. For example, should we allow for the neural_parameters to be perturbed during the tuning phase, rather than only during the topological mutation phase? What percentage of tuning should be dedicated to learning parameters and what percentage to synaptic weights? This of course can be tested, and benchmarked, and in general deduced through experimentation. After it has been decided on what and when to tune with regards to learning rules, there is still a problem with regards to the parameter and synaptic weight backup during the tuning phase. The main problem of this section is with regards to this dilemma, the dilemma of the backup process of the tuned weights.

Consider a neuron that has plasticity enabled, no matter what plasticity function it's using. The following scenario occurs when the neuron is perturbed:

1. The neuron receives a perturbation request.
2. Neuron selects random synaptic weights, weight_parameters, or even neural_parameters (though we do not allow for neural_parameters perturbation during the tuning phase, yet).
3. Then the agent gets re-evaluated, and **IF:**
    4. Perturbed agent has a higher fitness: the neuron backups its *current* weights/parameters.
    5. Perturbed agent has a lower fitness: the neuron restores its previous backed up weights/parameters.

There is a problem with step 4. Because by the time it's time to backup the synaptic weights, they have already changed from what they original started with during the evaluation, since they have adapted and learned due to their plasticity function. So we would not be backing up the synaptic weights of the agent that achieved the higher fitness score, but instead we would be backing up the learned and adapted agent with its adapted synaptic weights.

The fact that the perturbed agent, or topologically mutated agent, is not simply a perturbed genotype on which its parent is based, but instead is based on the genotype which has resulted from its parent's experience (due to the parent having changed based on its learning rule, before its genotype was backed up), means that the process is now based on Lamarckian evolution, rather than the biologically correct Darwinian. The definition of Lamarckian Evolution is based on the idea that an organism can pass on to its offspring the characteristics that it has acquired and learned during its lifetime (evaluation), all its knowledge and learned skills. Since plasticity affects the agent's neural patterns, synaptic weights... all of which are defined and written back to the agent's genotype, and the offspring is a mutated version of that genotype, the offspring thus in effect will to some extent inherit

the agent's adapted genotype, and not the original genotype with which the parent started when it was being evaluated.

When the agent backs up its synaptic weights after it has been evaluated for fitness, the agent uses Lamarckian evolution, because its experience, what it has learned during its evaluation (and what it has learned is reflected in how the synaptic weights changed due to the used plasticity learning rule), is written to its genome, and it is this learned agent that gets perturbed. The cleanest way to solve this problem, and have control of whether we use Lamarckian or the biologically correct Darwinian evolution, is to add a new parameter to the agent, the *darwinian*/*lamarckian* flag.

Darwinian vs. Lamarckian evolution, particularly in ALife simulations, could lead to interesting possibilities. When using Lamarckian evolution, and for example applying our neuroevolutionary system to an ALife problem, the agent's experience gained from interacting with the simulated environment, would be passed on to its offspring, and perturbed during the tuning phase. The perturbed organism (during the tuning phase, belonging to the same evaluation) would re-experience the interaction with the environment, and if it was even more successful, it would be backed up with its new experience (which means that the organism has now experienced and learned in the environment twice, since through plasticity the environment has affected its synaptic weights twice...). If the perturbed agent is less fit, then the previous agent, with its memories and synaptic weight combination, is reverted to, and re-perturbed. If we set the max_attempts counter to 1, then it will be genetic rather than a memetic based neuroevolutionary system. But again, when Lamarckian evolution is allowed, the memories of the parent are passed on to its offspring... A number of papers have researched the usefulness and efficiency of Darwinian Vs. Lamarkian evolution [4,5,6,7]. The results vary, and so adding a *heredity* flag to the agent will allow us to experiment and use both if we want to. We could then switch between the two heredity approaches (Darwinian or Lamarckian) easily, or perhaps even allow the hereditary flag to flip between the two during the topological mutation phase through some new topological mutation operator, letting the evolutionary process decide what suits the assigned problem best.

To implement the proper synaptic weight updating method to reflect the decided on hereditary approach during the tuning phase, we will need to add minor updates to the records.hrl file, the exoself, the neuron, and the genotype modules. In the records.hrl, we have to update the *agent* record by adding the *heredity_type* flag to it, and modifying the constraint record by adding the *heredity_types* element to it. The agent's *heredity_type* element will simply store a tag, an atom which can either be : *darwinian* or *lamarckian*. The constraint's heredity_types element will be a list of heredity_type tags. This list can either contain just a single tag, *'darwinian'* or *'lamarckian'* for example, or it could contain both. If both atoms are present in the heredity_types list, then during the creation of the seed population, some agents will use the darwinian method of passing on their heredi-

tary information, and others will use a lamarckian approach. It would be interesting to see which of the two would have an advantage, or be able to evolve faster, and during what stages of evolution and in which problems...

After updating the 2 records in records.hrl, we have to make a small update to the genotype module. In the genotype module we update the construct_Agent/3 function, and set the agent's heredity_type to one of the available heredity types in the constraint's heredity_types list. We do this by adding the following line when setting the agent's record: *heredity_type = random_element (SpecCon#constraint.heredity_types)*. We then update the exoself module, by modifying the link_Neurons/2 function to link_Neurons/3 function, and pass to it the agent's heredity_type parameter, the parameter which is then forwarded to each spawned neuron.

With this done, we make the final and main source modification, which is all contained within the neuron module. To allow for Darwinian based heredity in the presence of learning and plastic neurons, we need to keep track of two states of the input_pidps:

1. The input_pidps that are currently effective and represent the neuron's processing dynamics, which is the input_pidps_current.
2. A second input_pidps list, which represents the state of input_pidps right after perturbation, before the synaptic weights are affected by the neuron's plasticity function.

We can call this new list the *input_pidps_bl*, where *bl* stands for Before Learning.

When a neuron is requested to perturb its synaptic weights, right after the weights are perturbed, we want to save this new input_pidps list, before plasticity gets a chance to modify the synaptic weights. Thus, whereas before we stored the Perturbed_PIdPs in input_pidps_current, we now also save it to input_pidps_bl. Afterwards, the neuron can process the input signals using its input_pidps_current, and its learning rule can affect the input_pidps_current list. But input_pidps_bl will remain unchanged.

When a neuron is sent the weight_backup message, it is here that heredity_type plays its role. When it's *darwinian*, the neuron saves the *input_pidps_bl* to input_pidps_backup, instead of the input_pidps_current which could have been modified by some learning rule by this point. On the other hand, when the heredity_type is *lamarckian*, the neuron saves the input_pidps_current to input_pidps_backup. The input_pidps_current represents the synaptic weights that could have been updated if the neuron allows for plasticity, and thus the input_pidps_backup will then contain not the initial states of the synaptic weight list with which the neuron started, but the state of the synaptic weights after the neuron has experienced, processed, and had its synaptic weights modified by its learning rule. Using this logic we add to the neuron's state the element input_pidps_bl, and update the loop/6 function, as shown in Listing-15.18.

Listing-15.18 The neuron's loop/6 function which can use both, Darwinian and Lamarckian inheritance.

```
loop(S,ExoSelf_PId,[ok],[ok],SIAcc,MIAcc)->
    PF = S#state.pf,
    AF = S#state.af,
    AggrF = S#state.aggrf,
    {PFName,PFParameters} = PF,
    Ordered_SIAcc = lists:reverse(SIAcc),
    SI_PIdPs = S#state.si_pidps_current,
    SAggregation_Product = signal_aggregator:AggrF(Ordered_SIAcc,SI_PIdPs),
    SOutput = functions:AF(SAggregation_Product),
    Output_PIds = S#state.output_pids,
    [Output_PId ! {self(),forward,[SOutput]} || Output_PId <- Output_PIds],
    Ordered_MIAcc = lists:reverse(MIAcc),
    MI_PIdPs = S#state.mi_pidps_current,
    MAggregation_Product = signal_aggregator:dot_product(Ordered_MIAcc,MI_PIdPs),
    MOutput = functions:tanh(MAggregation_Product),
    U_SI_PIdPs = plasticity:PFName([MOutput|PFParameters],Ordered_SIAcc,SI_PIdPs,
SOutput),
    U_S=S#state{
            si_pidps_current = U_SI_PIdPs
    },
    SI_PIds = S#state.si_pids,
    MI_PIds = S#state.mi_pids,
    loop(U_S,ExoSelf_PId,SI_PIds,MI_PIds,[],[]);
loop(S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc)->
    receive
            {SI_PId,forward,Input}->
                    loop(S,ExoSelf_PId,SI_PIds,[MI_PId|MI_PIds],[{SI_PId,Input}|SIAcc],
MIAcc);
            {MI_PId,forward,Input}->

    loop(S,ExoSelf_PId,[SI_PId|SI_PIds],MI_PIds,SIAcc,[{MI_PId,Input}|MIAcc]);
            {ExoSelf_PId,weight_backup}->
                    U_S=case S#state.heredity_type of
                            darwinian ->
                                    S#state{
                                            si_pidps_backup=S#state.si_pidps_bl,
                                            mi_pidps_backup=S#state.mi_pidps_current
                                    };
                            lamarckian ->
                                    S#state{
                                            si_pidps_backup=S#state.si_pidps_current,
                                            mi_pidps_backup=S#state.mi_pidps_current
```

```
                        }
                end,
                loop(U_S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc);
        {ExoSelf_PId,weight_restore}->
                U_S = S#state{
                        si_pidps_bl=S#state.si_pidps_backup,
                        si_pidps_current=S#state.si_pidps_backup,
                        mi_pidps_current=S#state.mi_pidps_backup
                },
                loop(U_S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc);
        {ExoSelf_PId,weight_perturb,Spread}->
                Perturbed_SIPIdPs=perturb_IPIdPs(Spread,S#state.si_pidps_backup),
                Perturbed_MIPIdPs=perturb_IPIdPs(Spread,S#state.mi_pidps_backup),
                U_S=S#state{
                        si_pidps_bl=Perturbed_SIPIdPs,
                        si_pidps_current=Perturbed_SIPIdPs,
                        mi_pidps_current=Perturbed_MIPIdPs
                },
                loop(U_S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc);
        {ExoSelf_PId,reset_prep}->
                neuron:flush_buffer(),
                ExoSelf_PId ! {self(),ready},
                RO_PIds = S#state.ro_pids,
                receive
                        {ExoSelf_PId, reset}->
                                fanout(RO_PIds,{self(),forward,[?RO_SIGNAL]})
                end,
                loop(S,ExoSelf_PId,S#state.si_pids,S#state.mi_pids,[],[]);
        {ExoSelf_PId,get_backup}->
                NId = S#state.id,
                ExoSelf_PId ! {self(),NId,S#state.si_pidps_backup,S#state.mi_pidps_backup},
                loop(S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc);
        {ExoSelf_PId,terminate}->
                io:format("Neuron:~p is terminating.~n",[self()])
        after 10000 ->
                io:format("neuron:~p stuck.~n",[S#state.id])
end.
```

With this modification, our neuroevolutionary system can be used with Darwinian and Lamarckian based heredity. If we start the population_monitor process with a constraint where the agents are allowed to have neurons with plasticity, and set the heredity_types to either [lamarckian] or [darwinian,lamarckian], then some of the agents will have plasticity and be able to use the Lamarckian inheritance.

We can next add a simple mutation operator which works similarly to the way the mutation operators of other evolutionary strategy parameters work. We simply check whether there are any other heredity types in the constraint's *heredity_types* list, if there are, we change the currently used one to a new one, randomly chosen from the list. If there are no others, then the mutation operator exits with an error, without wasting the topological mutation attempt. This simple *mutate_heredity_type* mutation operator implementation is shown in Listing-15.19.

Listing-15.19 The implementation of the genome_mutator:mutate_heredity_type/1 mutation operator.

```
mutate_heredity_type(Agent_Id)->
    A = genotype:read({agent,Agent_Id}),
    case (A#agent.constraint)#constraint.heredity_types -- [A#agent.heredity_type] of
            [] ->
                    exit("********ERROR:mutate_heredity_type/1:: Nothing to mutate, only a
single function available.");
            Heredity_Type_Pool->
                    New_HT = lists:nth(random:uniform(length(Heredity_Type_Pool)),
Heredity_Type_Pool),
                    U_A = A#agent{heredity_type = New_HT},
                    genotype:write(U_A)
    end.
%mutate_heredity_type/1 function checks if there are any other heredity types in the agent's
constraint record. If any other than the one currently used by the agent is present, the agent ex-
changes the heredity type it currently uses for a random one from the remaining list. If no other
heredity types are available, the mutation operator exits with an error, and the
neuroevolutionary system tries another mutation operator.
```

Since this particular neuroevolutionary feature is part of the evolutionary strategies, we add it to the evolutionary strategy mutator list, which we created earlier:

```
-define(ES_MUTATORS,[
    mutate_tuning_selection,
    mutate_tuning_duration,
    mutate_tuning_annealing,
    mutate_tot_topological_mutations,
    mutate_heredity_type
]).
```

With this final modification, our neuroevolutionary system can now fully employ plasticity, and two types of heredity inheritance methods. We now finally compile, and test our updated system on the T-Maze Navigation problem we developed in the previous chapter.

15.6 Compiling & Testing

Our TWEANN system can now evolve NNs with plasticity, which means the evolved agents do not simply have an evolved response/reflex to sensory signals, but can also change, adapt, learn, modify their strategies as they interact with the ever changing and dynamic world. Having added this feature, and having created the T-Maze Navigation problem which requires the NN to change its strategy as it interacts with the environment, we can now test the various plasticity rules to see whether the agents will be able to achieve a fitness of 149.2, a fitness score achieved when the agent can gather the highest reward located in the right corner, and then when sensing that the reward is now not 1 but 0.2 in the right corner, start moving to the left corner to continue gathering the highest reward.

Having so significantly modified the records and the various modules, we reset the mnesia database after recompiling the modules. To do this, we first execute polis:sync(), then polis:reset(), and then finally polis:start() to startup the polis process. We have created numerous plasticity learning rules: [hebbian_w, hebbian, ojas_w, ojas, self_modulationV1, self_modulationV2, self_modulationV3, self_modulationV4, self_modulationV5, self_modulationV6, neuromodulation], too many to show the console printouts of. Here I will show you the results I achieved while benchmarking the *hebbian_w* and the *hebbian* learning rules, and I highly recommend testing the other learning rules by using the provided source code in the supplementary material.

To run the benchmarks, we first modify the ?INIT_CONSTRAINTS in the benchmarker module, setting the constraint's parameter: *neural_pfns,* to one of these plasticity rules for every benchmark. We can leave the *evaluations_limit* in the *pmp* record as 5000, but in the experiments I've performed, I set the population limit to 20 rather than 10, to allow for a greater diversity. The following are the results I achieved when running the experiments for the hebbian_w and the hebbian plasticity based benchmarks:

T-Maze Navigation with neural_pfns=[hebbian_w]:

```
Graph:{graph,discrete_tmaze,
        [1.1185328852434115,1.1619749686158354,1.1524569668377718,
         1.125571504518873,1.1289114832535887,1.1493175172780439,
         1.136998936735779,1.151456292245766,1.1340011357153639,
         1.1299993522129745],
        [0.0726690757747553,0.08603433346506212,0.07855604082593783,
         0.10142838037124464,0.07396159578145513,0.10671412852082847,
         0.07508707481514428,0.09451139923220694,0.10140517337683815,
         0.07774940615923569],
        [91.76556804891021,101.28562704890575,111.38602998360439,
         110.65857974481669,110.16398032961199,111.09056977671462,
         110.92899944938112,110.89051253132838,115.36595268212,
```

```
        111.07567142455073],
       [14.533256849468248,13.058657299854085,10.728855341054617,
        10.993110357580642,10.14374645989871,8.753610288273324,
        8.392536182954592,7.795296190771122,5.718415463002469,
        8.367092075873826],
       [122.00000000000001,122.00000000000001,148.4,149.2,149.2,149.2,
        149.2,149.2,149.2,149.2],
       [10.000000000000115,10.000000000000115,10.000000000000115,
        10.000000000000115,10.000000000000115,10.000000000000115,
        10.000000000000115,10.000000000000115,10.000000000000115,
        10.000000000000115],
       [11.45,14.3,15.3,15.8,15.3,16.15,16.15,15.55,15.95,15.7],
       [1.5321553446044565,2.451530134426253,2.1702534414210706,
        2.541653005427767,2.2825424421026654,2.7253440149823285,
        2.127792283095321,2.0118399538730714,2.246664193866097,
        2.0273134932713295],
       [500.0,500.0,500.0,500.0,500.0,500.0,500.0,500.0,500.0,500.0],
       []}
Tot Evaluations Avg:5172.95 Std:103.65301491032471
```

The boldfaced list shows the maximum achieved scores from all the evolutionary runs, and this time through plasticity, the score of 149.2 was achieved, implying our TWEANN's ability to solve the T-Maze navigation problem in under 2000 evaluationss (by the 4th of the 500th evaluations set).

T-Maze Navigation with neural_pfns=[hebbian]:

```
Graph:{graph,discrete_tmaze,
       [1.1349113313586998,1.1720830155097892,1.1280659983291563,
        1.1155462519936203,1.1394258373205741,1.1293439592742998,
        1.1421323920317727,1.1734812130593864,1.1750255550524766,
        1.2243932469319467],
       [0.07930932911768754,0.07243567080038446,0.0632406890972406,
        0.05913247338612391,0.0790334112982764 2,0.07030745338352402,
        0.09215871275247499,0.09666623776054033,0.1597898002580627,
        0.2447504142533042],
       [90.66616594516601,97.25899378881999,104.36751796157071,
        105.0985582137162,106.70360792131855,108.09892415530814,
        108.23839098414494,109.28814527629243,108.0643063975331,
        111.0103593241125],
       [15.044059269853784,13.919179099169385,10.613477213673535,
        13.557400867791436,13.380234103652047,12.413686820724935,
        11.936102929326337,11.580780191261242,12.636714964991167,
        12.816711475442705],
       [122.00000000000001,147.8,145.60000000000002,149.2,149.2,149.2,
```

```
        149.2,149.2,149.2,149.2],
       [10.000000000000115,10.000000000000115,10.000000000000115,
        10.000000000000115,10.000000000000115,10.000000000000115,
        10.000000000000115,10.000000000000115,10.000000000000115,
        10.000000000000115],
       [11.05,12.2,12.3,12.85,13.35,14.25,14.35,15.3,15.4,14.9],
       [1.6271140095272978,2.6381811916545836,2.215851980616034,
        1.7399712641305316,1.7399712641305318,2.2332711434127295,
        1.9817921182606415,2.0760539492026697,1.9078784028338913,
        2.046948949045872],
       [500.0,500.0,500.0,500.0,500.0,500.0,500.0,500.0,500.0,500.0],
       []}
Tot Evaluations Avg:5145.65 Std:91.87234349900953
```

In this case, our TWEANN again was able to solve the T-Maze problem. Plastic NN based agents do indeed have the ability to solve the T-Maze problem which requires the agents to change their strategy as they interact with the maze which changes midway. Our TWEANN is now able to evolve such plastic NN based agents, our TWEANN can now evolve agents that can learn new things as they interact with the environment, that can change their behavioral strategies based on their experience within the environment.

15.7 Summary & Discussion

Though we have tested only two of the numerous plasticity learning rules we've implemented, they both produced success. In both cases our TWEANN platform has been able to evolve NN based agents capable of solving the T-Maze problem, which was not solvable by our TWEANN in the previous chapter without plasticity. Thus we have successfully tested our plasticity rule implementations, and the new performance capabilities of our TWEANN. Outside this text I have tested the learning rules which were not tested above, and they are also capable of solving this problem, with varying performance levels. All of this without us having even optimized our algorithms yet.

With this benchmark complete, we have now finished developing numerous plasticity learning rules, implementing the said algorithms, and then benchmarking their performance. Our TWEANN system has finally been able to solve the T-Maze problem which requires the agents to change their strategy. Our TWEANN platform can now evolve not only complex topologies, but NN systems which can learn and adapt. Our system can now evolve thinking neural network based agents. There is nothing stopping us from producing more complex and more biologically faithful plasticity based learning rules, which would further improve the

capabilities and potential of the types of neural networks our TWEANN system can evolve.

With the plasticity now added, our next step is to add a completely different NN encoding, and thus further advance our TWEANN system. In the next chapter we will allow our TWEANN platform to evolve not only the standard encoded NN based agents we've been using up to this point, but also the new indirect encoded type of NN systems, the substrate encoded NN based systems.

15.8 References

[1] Oja E (1982) A Simplified Neuron model as a Principal Component Analyzer. Journal of Mathematical Biology 15, 267-273.

[2] Soltoggio A, Bullinaria JA, Mattiussi C, Durr P, Floreano D (2008) Evolutionary Advantages of Neuromodulated Plasticity in Dynamic, Reward-based Scenarios. Artificial Life 2, 569-576.

[3] Blynel J, Floreano D (2003) Exploring the T-maze: Evolving Learning-Like Robot Behaviors using CTRNNs. Applications of evolutionary computing 2611, 173-176.

[4] Whitley LD, Gordon VS, Mathias KE (1994) Lamarckian Evolution, The Baldwin Effect and Function Optimization. In Parallel Problem Solving From Nature - PPSN III, Y. Davidor and H. P. Schwefel, eds. (Springer), pp. 6-15.

[5] Julstrom BA (1999) Comparing Darwinian, Baldwinian, and Lamarckian Search in a Genetic Algorithm For The 4-Cycle Problem. In Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference, S. Brave and A. S. Wu, eds., pp. 134-138.

[6] Castillo PA, Arenas MG, Castellano JG, Merelo JJ, Prieto A, Rivas V, Romero G (2006) Lamarckian Evolution and the Baldwin Effect in Evolutionary Neural Networks. CoRR abs/cs/060, 5.

[7] Esparcia-Alcazar A, Sharman K (1999) Phenotype Plasticity in Genetic Programming: A Comparison of Darwinian and Lamarckian Inheritance Schemes. In Genetic Programming Proceedings of EuroGP99, R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, eds. (Springer-Verlag), pp. 49-64.