Chapter 6 Developing a Feed Forward Neural Network

Abstract In this chapter we discuss how a single artificial neuron processes signals, and how to simulate it. We then develop a single artificial neuron and test its functionality. Having discussed and developed a single neuron, we decide on the NN architecture we will implement, and then develop a genotype constructor, and a mapper from genotype to phenotype. Finally, we then ensure that that our simple NN system works by using a simple sensor and actuator attached to the NN to test its sense-think-act ability.

As we discussed in an earlier chapter, Neural Networks (NN) are directed graphs composed of simple processing elements as shown in Figure-6.1. Every vertex in such a directed graph is a Neuron, every edge is an outgoing axon and a path along which the neuron sends information to other Neurons. A NN has an input layer which is a set of neurons that receive signals from sensors, and an output layer which is a set of neurons that connect to actuators. In a general NN system the sensors can be anything, from cameras, to programs that read from a database and pass that data to the neurons. The Actuators too can range from functions which control motors, to simple programs which print the output signals to the screen. Every neuron processes its incoming signals, produces an output signal, and passes it on to other neurons.

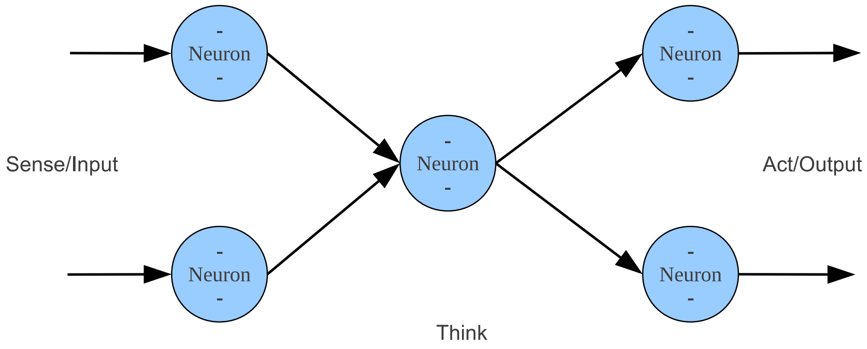


Fig. 6.1 A simple Neural Network.

Whether the NN does something intelligent or useful is based on its topology and parameters. The method of modifying the NN topology and parameters to make it do something useful, is the task of its learning algorithm. A learning algorithm can be supervised, like in the case of the error back propagation learning algorithm, or it can be unsupervised like in the evolutionary or reinforcement learning algorithms. In a supervised learning algorithm the outputs of the NN need to

DOI 10.1007/978-1-4614-4463-3_6, © Springer Science+Business Media New York 2013

be known in advance, such that corrections can be given to the NN based on the differences in its produced outputs and the correct outputs. Once we have minimized the differences between the answers we want and the answers the NN gives, we apply the NN to a new set of data, to another problem in the same field but one which the NN has not encountered during its training. In the case of unsupervised learning, it is only important to be able to tell whether one NN system performs better than another. There is no need to know exactly how the problem should be solved, the NNs will try to figure that out for themselves; the researcher only needs to choose the neural networks that produce better results over those that do not. We will develop these types of systems in future sections.

In this chapter we will learn how to program a static neural network system whose topological and parametric properties are specified during its creation, and are not changed during training. We will develop a genotype encoding for a simple monolithic Neural Network, and then we'll create a mapper program which converts the NN genotype to its phenotypic representation. The process of modifying these weights, parameters, and the NN topology is the job of a learning algorithm, the subject that we will cover in the chapters that follow.

In the following sections when we discuss genotypes and phenotypes, we mean their standard definitions: a genotype is the organism's full hereditary information, which is passed to offspring in mutated or unchanged form, and the phenotype is the organism's actual observed properties, its morphology and behavior. The process of mapping a genotypical representation of the organism to the phenotypical one is done through a process called development, to which we also will refer to as: mapping. A genotype of the organism is the form in which we store it in our database, on the other hand its phenotype is its representation and behavior when the organism, a NN in our case, is live and functioning. In the NN system that we build in this chapter, the genotype will be a list of tuples, and the phenotype a graph of interconnected processes sending and receiving messages from one another.

********Note********
The encoding of a genotype itself can either be direct, or indirect. A direct encoding is one in which the genotype encodes every topological and parametric aspect of the NN phenotype in a one to one manner, the genotype and the phenotype can be considered one and the same. An indirect encoding applies a set of programs, or functions to the genotype, through which the phenotype is developed. This development process can be highly complex and stochastic in nature which takes into consideration the environmental factors during the time of development, and producing a one too many mapping from a genotype to the phenotype. An example of a direct encoding is that of a bit string which maps to a colored strip in which the 0s are directly converted to white sections and 1s to black. An example of an indirect encoding is the case of DNA, where the development from the genotype to a phenotype is a multi-stage process, with complex interactions between the developing organism and the environment it is in.
****

We will now slowly build up a NN system, from a single neuron, to a fully functional feed forward neural network. In the next section we take our first step and develop an artificial neuron using Erlang.

6.1 Simulating A Neuron

Let us again briefly review the representation and functionality of a single artificial neuron, as shown in Figure-6.2. A neuron is but a simple processing element which accepts input signals, weighs the importance of each signal by multiplying it by a weight associated with it, adds a bias to the result, applies an activation function to this sum, and then forwards the result to other elements it is connected to. As an example, assume we have a list of input signals to the neuron: [I1,I2,I3,I4], this input is represented as a vector composed of 4 elements. The neuron then must have a list of weights, one weight for every incoming signal: [W1,W2,W3,W4]. We weigh each signal with its weight by taking a dot product of the input vector and the weight vector as follows: Dot_Product = I1*W1 + I2*W2 + I3*W3 + I4*W4. If the neuron also has a threshold value or bias, we simply add this bias value to the Dot_Product. Finally, we apply the activation

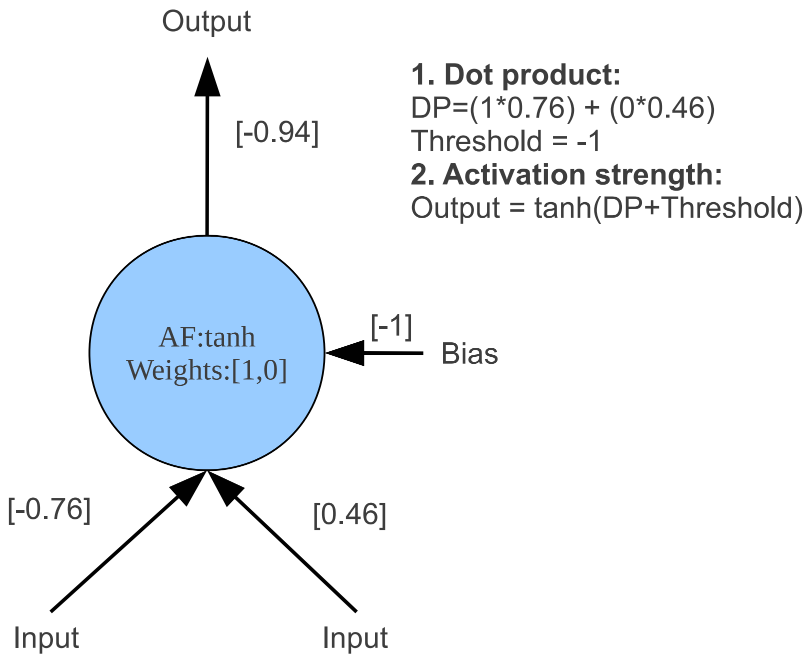


Fig. 6.2 An artificial Neuron.

function to the dot product to produce the final output of the neuron: Output = Activation_Function(Dot_Product), and for a neuron that also has a bias: Output = Activation_Function(Dot_Product + Bias). A bias is an extra floating point parameter not associated with any particular incoming signal, and it adds a level of tunable asymmetry to the activation function.

Mathematically, the neuron that uses a set of weights and a bias is equivalent to a neuron that accepts an "extended input vector" and uses an "extended weight vector" to weigh the signals. An extended input vector has "1" appended to the input vector and an extended weight vector has the bias appended to the weight vector. Using the extended vectors, we then take a single dot product as follows: [I1,I2,I3,I4,1]dot[W1,W2,W3,W4,Bias]= (I1*W1) +(I2*W2) +(I3*W3) +(I4*W4) +(1*Bias), which is equal to the dot product of the input and weight vector, plus the bias as before. Neurons that do not use a bias would simply not append the extension to the input, and thus produce the dot product without a bias value.

Lets simulate and test a very simple neuron, which we will represent using a process. The neuron will have a predetermined number of weights, 2, and it will include a bias. With 2 wights, this neuron can process input vectors of length 2. The activation function will be the standard sigmoid function, in our neuron it's approximated by the hyperbolic tangent (tanh) function included in the math module. The architecture of this neuron will be the same as in Figure-6.2.

In the following algorithm, we spawn a process to represent our Neuron, and register it so that we can send and receive signals from it. We use a simple remote procedure call function called 'sense' to send signals to the registered neuron, and then receive the neuron's output.

```
simple_neuron.erl
-module(simple_neuron).
-compile(export_all).

create()->
    Weights = [random:uniform()-0.5,random:uniform()-0.5,random:uniform()-0.5],
    register(neuron, spawn(?MODULE,loop,[Weights])).
%The create function spawns a single neuron, where the weights and the bias are generated
randomly to be between -0.5 and 0.5.

loop(Weights) ->
    receive
            {From, Input} ->
                    io:format("****Processing****~n Input:~p~n Using
Weights:~p~n",[Input,Weights]),
                    Dot_Product = dot(Input,Weights,0),
                    Output = [math:tanh(Dot_Product)],
                    From ! {result,Output},
                    loop(Weights)
    end.
%The spawned neuron process accepts an input vector, prints it and the weight vector to the
screen, calculates the output, and then sends the output to the contacting process. The output is
also a vector of length one.

    dot([I|Input],[W|Weights],Acc) ->
            dot(Input,Weights,I*W+Acc);
    dot([],[Bias],Acc)->
            Acc + Bias.
%The dot product function that we use works on the assumption that the bias is incorporated in-
to the weight list as the last value in that list. After calculating the dot product, the input list will
empty out while the weight list will still have the single bias value remaining, which we then
add to the accumulator.

sense(Signal)->
    case is_list(Signal) and (length(Signal) == 2) of
            true->
                    neuron ! {self(),Signal},
                    receive
                            {result,Output}->
                            io:format(" Output: ~p~n",[Output])
                    end;
            false->
                    io:format("The Signal must be a list of length 2~n")
    end.
%We use the sense function to contact the neuron and send it an input vector. The sense func-
tion ensures that the signal we are sending is a vector of length 2.
```

Now let’s compile and test our module:

```
1> c(simple_neuron).
{ok,simple_neuron}
2> simple_neuron:create().
true.
3> simple_neuron:sense([1,2]).
****Processing****
 Input:[1,2]
 Using Weights:[0.44581636451986995,0.0014907142064750634, -0.18867324519560702]
 Output: [0.25441202264242263]
```

It works! We can expand this neuron further by letting it accept signals only from certain predetermined list of PIds, and then output the result not back to those same processes, but instead to another set of PIds. With such modifications this neuron could then be used as a fully functional processing element in a NN. In the next section we will build a single neuron neural network that uses such processing element.

6.2 A One Neuron Neural Network

Next we will create the simplest possible NN. Our NN topology will be composed of a single Neuron which receives a signal from a Sensor, calculates an output based on its weights and activation function, and then passes that output signal to the Actuator. This topology and architecture is shown in Figure-6.3. You will also notice that there is a 4th element called Cortex. This element is used to trigger the sensor to start producing sensory data, and it also contains the PIds of all the processes in the system so that it can be used to shut down the NN when we are done with it. Finally, this type of element can also be used as a supervisor of the NN, and play a role in the NN's synchronization with the learning algorithm. These features will become important when we start developing the more complex NN systems in the chapters that follow.

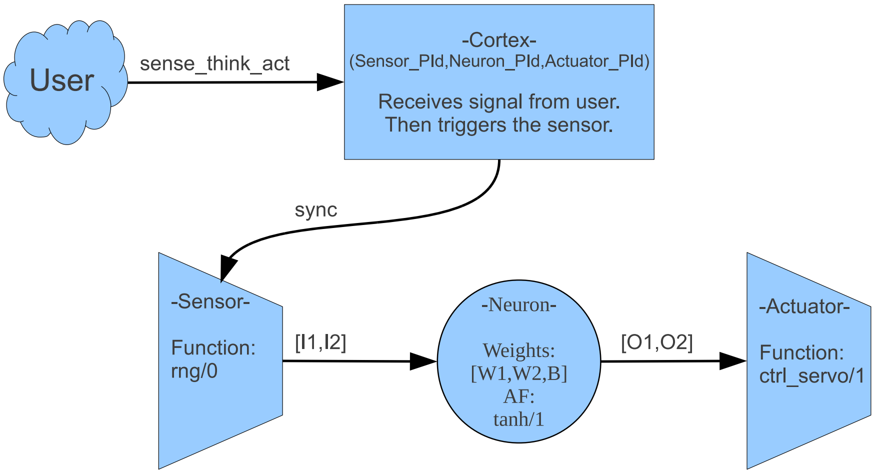


Fig. 6.3 One Neuron Neural Network.

To create this system, we will need to significantly modify the functions in our simple_neuron module, and add new features as shown in the following source code:

```
simplest_nn.erl
-module(simplest_nn).
-compile(export_all).

create() ->
   Weights = [random:uniform()-0.5,random:uniform()-0.5,random:uniform()-0.5],
   N_PId = spawn(?MODULE,neuron,[Weights,undefined,undefined]),
   S_PId = spawn(?MODULE,sensor,[N_PId]),
   A_PId = spawn(?MODULE,actuator,[N_PId]),
   N_PId ! {init,S_PId,A_PId},
   register(cortex,spawn(?MODULE,cortex,[S_PId,N_PId,A_PId])).
%The create function first generates 3 weights, with the 3rd weight being the Bias. The Neuron
is spawned first, and is then sent the PIds of the Sensor and Actuator that it's connected with.
Then the Cortex element is registered and provided with the PIds of all the elements in the NN
system.

neuron(Weights,S_PId,A_PId) ->
   receive
         {S_PId, forward, Input} ->
               io:format("****Thinking****~n Input:~p~n with
Weights:~p~n",[Input,Weights]),
               Dot_Product = dot(Input,Weights,0),
               Output = [math:tanh(Dot_Product)],
               A_PId ! {self(), forward, Output},
               neuron(Weights,S_PId,A_PId);
         {init, New_SPId, New_APId} ->
               neuron(Weights,New_SPId,New_APId);
         terminate ->
               ok
   end.
%After the neuron finishes setting its SPId and APId to that of the Sensor and Actuator respec-
tively, it starts waiting for the incoming signals. The neuron expects a vector of length 2 as in-
put, and as soon as the input arrives, the neuron processes the signal and passes the output vec-
tor to the outgoing APId.

   dot([I|Input],[W|Weights],Acc) ->
         dot(Input,Weights,I*W+Acc);
   dot([],[],Acc)->
```

```
		Acc;
	dot([],[Bias],Acc)->
		Acc + Bias.
%The dot function takes a dot product of two vectors, it can operate on a weight vector with
and without a bias. When there is no bias in the weight list, both the Input vector and the
Weight vector are of the same length. When Bias is present, then when the Input list empties
out, the Weights list still has 1 value remaining, its Bias.

sensor(N_PId) ->
	receive
		sync ->
			Sensory_Signal = [random:uniform(),random:uniform()],
			io:format("****Sensing****:~n Signal from the environment
~p~n",[Sensory_Signal]),
			N_PId ! {self(),forward,Sensory_Signal},
			sensor(N_PId);
		terminate ->
			ok
	end.
%The Sensor function waits to be triggered by the Cortex element, and then produces a random
vector of length 2, which it passes to the connected neuron. In a proper system the sensory sig-
nal would not be a random vector but instead would be produced by a function associated with
the sensor, a function that for example reads and vector-encodes a signal coming from a GPS
attached to a robot.

actuator(N_PId) ->
	receive
		{N_PId,forward,Control_Signal}->
			pts(Control_Signal),
			actuator(N_PId);
		terminate ->
			ok
	end.

	pts(Control_Signal)->
		io:format("****Acting****:~n Using:~p to act on environ-
ment.~n",[Control_Signal]).
%The Actuator function waits for a control signal coming from a Neuron. As soon as the signal
arrives, the actuator executes its function, pts/1, which prints the value to the screen.
```

```
cortex(Sensor_PId,Neuron_PId,Actuator_PId)->
    receive
            sense_think_act ->
                    Sensor_PId ! sync,
                    cortex(Sensor_PId,Neuron_PId,Actuator_PId);
            terminate ->
                    Sensor_PId ! terminate,
                    Neuron_PId ! terminate,
                    Actuator_PId ! terminate,
                    ok
    end.
%The Cortex function triggers the sensor to action when commanded by the user. This process
also has all the PIds of the elements in the NN system, so that it can terminate the whole system
when requested.
```

Lets compile and try out this system:

```
1>c(simplest_nn).
{ok,simplest_nn}
2>simplest_nn:create().
true
3> cortex ! sense_think_act.
****Sensing****:
 Signal from the environment [0.09230089279334841,0.4435846174457203]
sense_think_act
****Thinking****
 Input:[0.09230089279334841,0.4435846174457203]
 with Weights:[-0.4076991072066516,-0.05641538255427969,0.2230402056221108]
****Acting****:
 Using:[0.15902302907693572] to act on environment.
```

It works! But though this system does embody many important features of a real NN, it is still rather useless since it's composed of a single neuron, the sensor produces random data, and the NN has no learning algorithm so we can not teach it to do something useful. In the following sections we are going to design a NN system for which we can specify different starting topologies, for which we can specify sensors and actuators, and which will have the ability to learn to accomplish useful tasks.

6.3 Planning Our Neural Network System's Architecture

A standard Neural Network (NN) is a graph of interconnected Neurons, where every neuron can send and receive signals from other neurons and/or sensors and actuators. The simplest of NN architectures is that of a monolithic feed forward neural network (FFNN), as shown in Figure-6.4. In a FFNN, the signals only propagate in the forward direction, from sensors, through the neural layers, and finally reaching the actuators which use the output signals to act on the environment. In such a NN system there are no recursive or cyclical connections. After the Actuators have acted upon the environment, the sensors once again produce and send sensory signals to the neurons in the first layer, and the "Sense-Think-Act" cycle repeats.

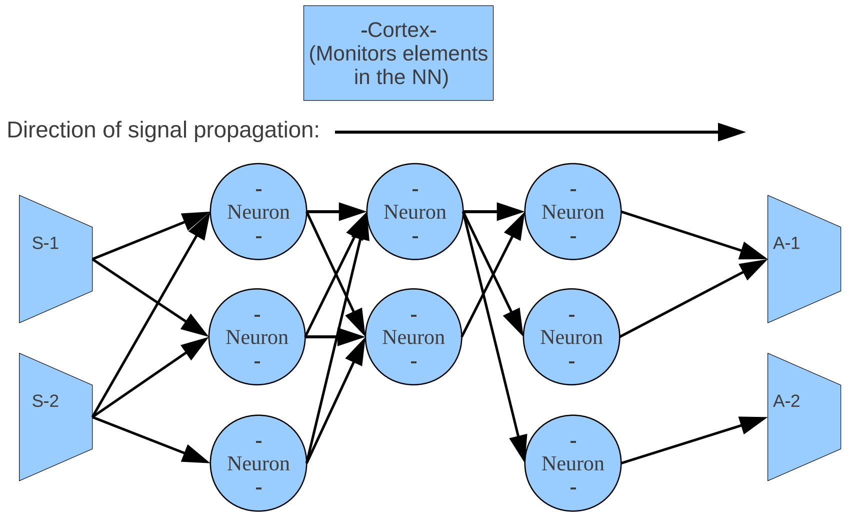

-Cortex-
(Monitors elements in the NN)
Direction of signal propagation:
S-1
S-2
- Neuron -
A-1
A-2

Fig. 6.4 A **Feed Forward Neural Network.**

Every neuron must be able to accept a vector input of length 1+, and produce a vector output of length 1. Since all neural inputs and outputs are in vector form, and the sensory signals sent from the sensors are also in vector form, the neurons neither need to know nor care whether the incoming signals are coming from other neurons or sensors. Let's take a closer look at the two types of connections that occur in a NN, the [neuron|sensor]-to-neuron and the neuron-to-actuator connection as shown in Figure-6.5.

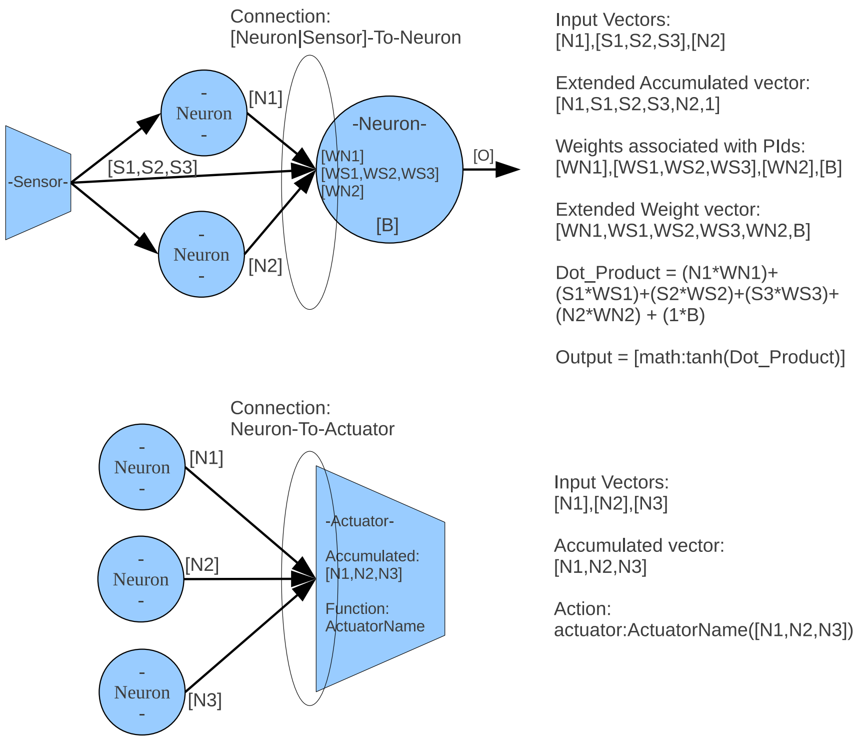


Fig. 6.5 Neuron/Sensor-To-Neuron & Neuron-To-Actuator connections.

Every input signal to a neuron is a list of values [I1...In], a vector of length 1 or greater. The neuron's output signal is also a vector, a list of length 1, [O]. Because each Neuron outputs a vector of length 1, the actuators accumulate the signals coming from the Neurons into properly ordered vectors of length 1+. The order of values in the vector is the same as the order of PIds in its fanin pid list. Once the actuator has finished gathering the signals coming from all the neurons connected to it, it uses the accumulated vector as a parameter to its actuation function.

Once all the neurons in the output layer have produced and forwarded their signals to actuators, the NN can start accepting new sensory inputs again (*Note* It is possible for a NN to process multiple sensory input vectors, one after the other, rather than one at a time and waiting until an output vector is produced before accepting a new wave of sensory vectors. This would be somewhat similar to the way a multi-stage pipeline in a CPU works, with every neural layer in the NN processing signals at the same time, as opposed to the processing of sensory vectors propagating from first to last layer, one set of sensory input vectors at a time.) This Sense-Think-Act cycle requires some synchronization, especially if a learning algorithm is also present. This synchronization will be done using the Cortex element we've briefly discussed earlier. We will recreate a more complex version of the Cortex program which will synchronize the sensors producing sensory signals, the actuators gathering the output vectors from the NN's output layer, and the learning algorithm modifying the weight parameters of the NN and allowing the system to learn.

Putting all this information and elements together, our Neural Network will function as follows: The sensor programs poll/request input signals from the environment, and then preprocess and fan out these sensory signals to the neurons in the first layer. Eventually the neurons in the output layer produce signals that are passed to the actuator program(s). Once an actuator program receives the signals from all the neurons it is connected from, it post-processes these signals and then acts upon the environment. A sensor program can be anything that produces signals, either by itself (random number generator) or as a result of interacting with the environment, like a camera, an intrusion detection system, or a program that simply reads from a database and passes those values to the NN for example. An

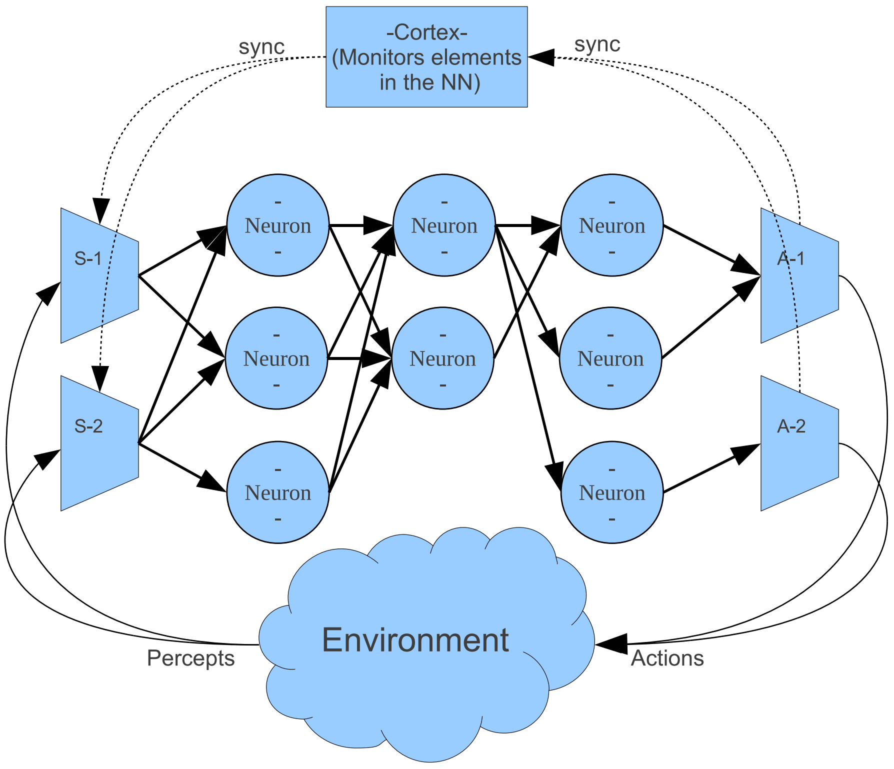


Fig. 6.6 All the elements of a NN system.

actuator program is any program that accepts signals and then acts upon the environment based on those signals. For example, a robot actuator steering program can accept a single floating point value, post process the value so that its range is from -1 to 1, and then execute the motor driver using this value as the parameter, where the sign and magnitude of the parameter designates which way to steer and how hard. Another example actuator is one that accepts signals from the NN, and then buys or sells a stock based on that signal, with a complementary sensor which reads the earlier price values of the same stock. This type of NN system architecture is visually represented in Figure-6.6.

A sensor, actuator, neuron, and the cortex are just 4 different types of processes that accept signals, process them, and execute some kind of element specific function. Lets discuss every one of these processes in detail, to see what information we might need to create them in their genotypic and phenotypic form.

Sensor: A sensor is any process that produces a vector signal that the NN then processes. This signal can be produced from the sensor interacting with the environment, for example the data coming from a camera, or from the sensor somehow generating the signal internally.

Actuator: An actuator is a process that accepts signals from the Neurons in the output layer, orders them into a vector, and then uses this vector to control some function that acts on the environment or even the NN itself. An actuator might have incoming connections from 3 Neurons, in which case it would then have to wait until all 3 of the neurons have sent it their output signals, accumulate these signals into a vector, and then use this vector as a parameter to its actuation function. The function could for example dictate the voltage signal to be sent to a servo that controls a robot's gripper.

Neuron: The neuron is a signal processing element. It accepts signals, accumulates them into an ordered vector, then processes this input vector to produce an output, and finally passes the output to other elements it is connected to. The Neuron never interacts with the environment directly, and even when it does receive signals and produces output signals, it does not know whether these input signals are coming from sensors or neurons, or whether it is sending its output signals to other neurons or actuators. All the neuron does is have a list of input PIds from which it expects to receive signals, a list of output PIds to which the neuron sends its output, a weight list correlated with the input PIds, and an activation function it applies to the dot product of the input vector and its weight vector. The neuron waits until it receives all the input signals, processes those signals, and then passes the output onwards.

Cortex: The cortex is a NN synchronizing element. It needs to know the PId of every sensor and actuator, so that it will know when all the actuators have received their control inputs, and that it's time for the sensors to again gather and fanout sensory data to the neurons in the input layer. At the same time, the Cortex element can also act as a supervisor of all the Neuron, Sensor, and Actuator elements in the NN system.

Now that we know how these elements should work and process signals, we need to come up with an encoding which can be used to store any type of NN topology in a database, or a flat file. This stored representation of the NN is its genotype. We should be able to specify the topology and the parameters of the NN within the genotype, and then generate from it a process based NN system, the phenotype. Using a genotype also allows us to train a NN to do something useful, and then save the updated and trained NN to a file for later use. Finally, once we decide to use an evolutionary learning algorithm, the NN genotypes are what the mutation operators will be applied to, and from what the mutated offspring will be generated.

In the next section we will develop a simple, human readable, and tuple based genotype encoding for our NN system. This type of encoding will be easy to understand, work with, and easy to encode and operate on using standard directional graph based functions. The use of such a direct way to store the genotype will also make it easy to think about it, and thus to advance, scale, and utilize it in the more advanced systems we'll develop in the future.

6.4 Developing a Genotype Representation

There are a number of ways to encode the genotype of a monolithic Neural Network (NN). Since NNs are directed graphs, we could simply use Erlang's digraph module. The digraph module in particular has functions with which to create Nodes/Neurons, Edges/Connections between the nodes, and even sub graphs, thus easily allowing us to develop modular topologies. Another simple way to encode the genotype is by representing the NN as a list of tuples, where every tuple is a record representing either a Neuron, Sensor, Actuator, or the Cortex element. Finally, we could also use a hash table, ets for example, instead of a simple list to store the tuples.

In every one of these cases, every element in the genotype is encoded as a human readable tuple. Our records will directly reflect the information that would be included and needed by every process in the phenotype. The 4 elements can be represented using the following records:

Sensor: **-record(sensor, {id, cx_id, name, vl, fanout_ids}).**

The sensor id has the following format: *{sensor, UniqueVal}*. *cx_id* is the Id of the Cortex element. 'name' is the name of the function the sensor executes to generate or acquire the sensory data, and *vl* is the vector length of the produced sensory signal. Finally, *fanout_ids* is a list of neuron ids to which the sensory data will be fanned out.

Actuator: **-record(actuator, {id, cx_id, name, vl, fanin_ids}).**

The actuator id has the following format: *{actuator, UniqueVal}*. *cx_id* is the the Id of the Cortex element. '*name*' is the name of the function the actuator executes to act upon the environment, with the function parameter being the vector it accumulates from the incoming neural signals. '*vl*' is the vector length of the accumulated actuation vector. Finally, the *fanin_ids* is a list of neuron ids which are connected to the actuator.

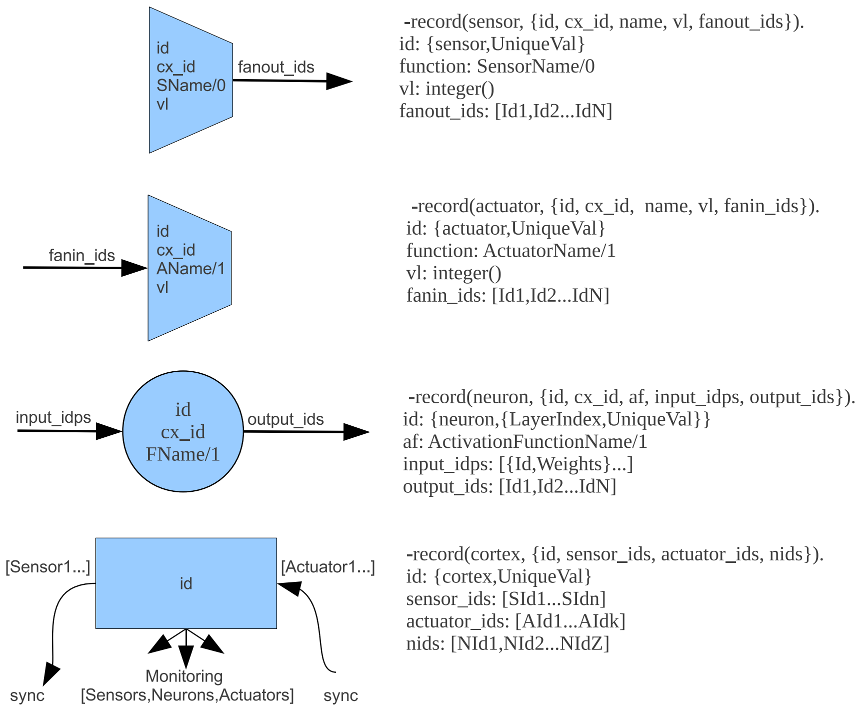
Neuron: **-record(neuron, {id, cx_id, af, input_idps, output_ids}).**

A neuron id uses the following format: *{neuron,{LayerIndex, UniqueVal}}*. *cx_id* is the the Id of the Cortex element. The activation function, *af*, is the name of the function the neuron uses on the extended dot product (dot product plus bias). The activation function that we will use in the simple NN we design in this chapter will be 'tanh', later we will extend the list of available activation functions our NNs can use. '*input_idps*' stands for Input Ids "Plus", which is a list of tuples as follows: *[{Id1,Weights1} ... {IdN,WeightsN},{bias,Val}]*. Each tuple is composed of the Id of the element that is connected to the neuron, and weights correlated with the input vector coming from the neuron with the listed Id. The last tuple in the input_idps is {bias,Val}, which is not associated with any incoming signal, and represents the Bias value. Finally, *output_ids* is a list of Ids to which the neuron will fanout its output signal.

Cortex: **-record(cortex, {id, sensor_ids, actuator_ids, nids}).**

The cortex Id has the following format: *{cortex, UniqueVal}*. '*sensor_ids*' is a list of sensor ids that produce and pass the sensory signals to the neurons in the input layer. '*actuator_ids*' is a list of actuator ids that the neural output layer is connected to. When the actuator is done affecting the environment, it sends the cortex a synchronization signal. After the cortex receives the sync signal from all the ids in its actuator_ids list, it triggers all the sensors in the sensor_ids list. Finally, *nids* is the list of all neuron ids in the NN.

Figure-6.7 shows the correlation between the tuples/records and the process based phenotypic representations to which they map. Using this record representation in our genotype allows us to easily and safely store all the information of our NN. We need only decide whether to use a digraph, a hash table, or a simple list to store the Genotype of a NN. Because we will be building a very simple Feed Forward Neural Network in this chapter, let us start by using a simple list. For the more advanced evolutionary NN systems that we'll build in the later chapters, we will switch to an ETS or a Digraph representation.



```
-record(sensor, {id, cx_id, name, vl, fanout_ids}).
id: {sensor,UniqueVal}
function: SensorName/0
vl: integer()
fanout_ids: [Id1,Id2...IdN]

-record(actuator, {id, cx_id,  name, vl, fanin_ids}).
id: {actuator,UniqueVal}
function: ActuatorName/1
vl: integer()
fanin_ids: [Id1,Id2...IdN]

-record(neuron, {id, cx_id, af, input_idps, output_ids}).
id: {neuron,{LayerIndex,UniqueVal}}
af: ActivationFunctionName/1
input_idps: [{Id,Weights}...]
output_ids: [Id1,Id2...IdN]

-record(cortex, {id, sensor_ids, actuator_ids, nids}).
id: {cortex,UniqueVal}
sensor_ids: [SId1...SIdn]
actuator_ids: [AId1...AIdk]
nids: [NId1,NId2...NIdZ]
```

Fig. 6.7 Record to process correlation.

In the next section we will develop a program which accepts high level specification parameters of the NN genotype we wish to construct, and which outputs the genotype represented as a list of tuples. We will then develop a mapping function which will use our NN genotype to create a process based phenotype, which is the actual NN system that senses, thinks, and takes action based on its sensory signals and neural processing.

6.5 Programming the Genotype Constructor

Now that we've decided on the necessary elements and their genotypic representation in our NN system, we need to create a program that accepts as input the high level NN specification parameters, and produces the genotype as output.

When creating a NN, we need to be able to specify the sensors it will use, the actuators it will use, and the general NN topology. The NN topology specification should state how many layers and how many neurons per layer the feed forward NN will have. Because we wish to keep this particular NN system very simple, we will only require that the genotype constructor is able to generate NNs with a single sensor and actuator. For the number of layers and layer densities of the NN, all the information can be contained in a single LayerDensities list as shown in Figure-6.8. Thus, our genotype constructor should be able to construct everything from a parameter list composed of a sensor name, an actuator name, and a LayerDensities list. The LayerDensities parameter will actually only specify the hidden layer densities, where the hidden LayerDensities are all the non output layer densities. The output layer density will be calculated from the vector length of the actuator. An empty HiddenLayerDensities list implies that the NN will only have a single neural layer, whose density is equal to the actuator's vector length.

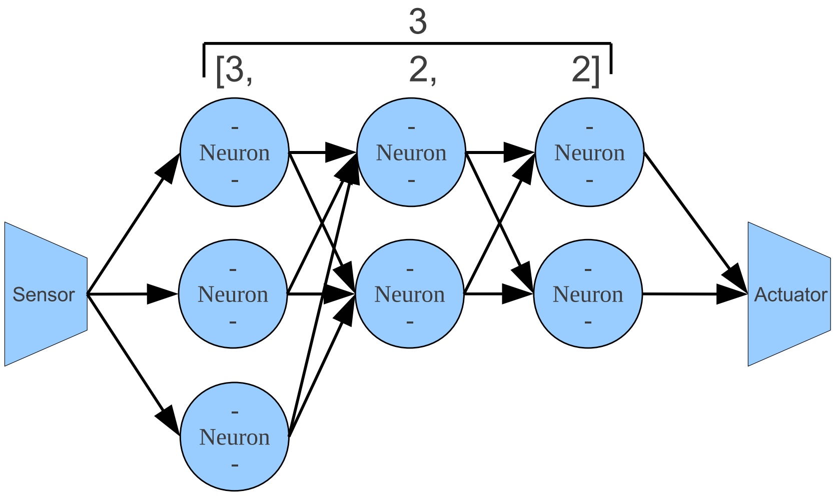


Fig. 6.8 A NN composed of 3 layers, with a [3, 2, 2] layer density pattern.

For example, a genotype creating program which accepts (SensorName,ActuatorName,[1,3]) as input, where the sensor vector length is 3 and the actuator vector length is 1, should produce a NN with 3 layers, whose output layer has 1 neuron, as shown in Figure-6.9. The input layer will have a single neuron which has 3 weights and a bias, so that the neurons in the first layer can process input vectors of length 3 coming from the sensor. The output layer has a single neuron, due to actuator's vl equaling 1.

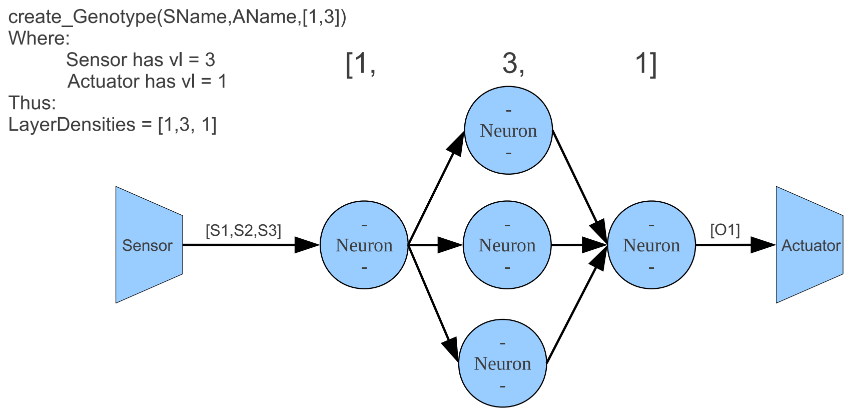


Fig. 6.9 Genotype with: LayerDensities == [1,3,1], and HiddenLayerDensities == [1,3].

We first create a file to contain the records representing each element we'll use:

```
records.hrl
-record(sensor, {id, cx_id, name, vl, fanout_ids}).
-record(actuator,{id, cx_id,  name, vl, fanin_ids}).
-record(neuron, {id, cx_id, af, input_idps, output_ids}).
-record(cortex, {id, sensor_ids, actuator_ids, nids}).
```

Now we develop an algorithm that constructs the genotype of a general feed forward NN based on the provided sensor name, actuator name, and the hidden layer densities parameter:

```
constructor.erl
-module(constructor).
-compile(export_all).
-include("records.hrl").

construct_Genotype(SensorName,ActuatorName,HiddenLayerDensities)->
    construct_Genotype(ffnn,SensorName,ActuatorName,HiddenLayerDensities).
construct_Genotype(FileName,SensorName,ActuatorName,HiddenLayerDensities)->
    S = create_Sensor(SensorName),
    A = create_Actuator(ActuatorName),
    Output_VL = A#actuator.vl,
    LayerDensities = lists:append(HiddenLayerDensities,[Output_VL]),
    Cx_Id = {cortex,generate_id()},

    Neurons = create_NeuroLayers(Cx_Id,S,A,LayerDensities),
    [Input_Layer|_] = Neurons,
    [Output_Layer|_] = lists:reverse(Neurons),
    FL_NIds = [N#neuron.id || N <- Input_Layer],
    LL_NIds = [N#neuron.id || N <-  Output_Layer],
    NIds = [N#neuron.id || N <- lists:flatten(Neurons)],
    Sensor = S#sensor{cx_id = Cx_Id, fanout_ids = FL_NIds},
    Actuator = A#actuator{cx_id=Cx_Id,fanin_ids = LL_NIds},
    Cortex = create_Cortex(Cx_Id,[S#sensor.id],[A#actuator.id],NIds),
    Genotype = lists:flatten([Cortex,Sensor,Actuator|Neurons]),
    {ok, File} = file:open(FileName, write),
    lists:foreach(fun(X) -> io:format(File, "~p.~n",[X]) end, Genotype),
    file:close(File).
%The construct_Genotype function accepts the name of the file to which we'll save the geno-
type, sensor name, actuator name, and the hidden layer density parameters. We have to generate
unique Ids for every sensor and actuator. The sensor and actuator names are used as input to the
create_Sensor and create_Actuator functions, which in turn generate the actual Sensor and Ac-
tuator representing tuples. We create unique Ids for sensors and actuators so that when in the
future a NN uses 2 or more sensors or actuators of the same type, we will be able to differenti-
ate between them using their Ids. After the Sensor and Actuator tuples are generated, we extract
the NN's input and output vector lengths from the sensor and actuator used by the system. The
Input_VL is then used to specify how many weights the neurons in the input layer will need,
and the Output_VL specifies how many neurons are in the output layer of the NN. After ap-
pending the HiddenLayerDensites to the now known number of neurons in the last layer to gen-
erate the full LayerDensities list, we use the create_NeuroLayers function to generate the Neu-
ron representing tuples. We then update the Sensor and Actuator records with proper fanin and
fanout ids from the freshly created Neuron tuples, compose the Cortex, and write the genotype
to file.

    create_Sensor(SensorName) ->
            case SensorName of
                    rng ->
                            #sensor{id={sensor,generate_id()},name=rng,vl=2};
                    _ ->
                            exit("System does not yet support a sensor by the
name:~p.",[SensorName])
            end.

    create_Actuator(ActuatorName) ->
            case ActuatorName of
                    pts ->
                            #actuator{id={actuator,generate_id()},name=pts,vl=1};
                    _ ->
                            exit("System does not yet support an actuator by the
name:~p.",[ActuatorName])
            end.
%Every sensor and actuator uses some kind of function associated with it, a function that either
polls the environment for sensory signals (in the case of a sensor) or acts upon the environment
(in the case of an actuator). It is the function that we need to define and program before it is
```

used, and the name of the function is the same as the name of the sensor or actuator itself. For example, the create_Sensor/1 has specified only the rng sensor, because that is the only sensor function we've finished developing. The rng function has its own vl specification, which will determine the number of weights that a neuron will need to allocate if it is to accept this sensor's output vector. The same principles apply to the create_Actuator function. Both, create_Sensor and create_Actuator function, given the name of the sensor or actuator, will return a record with all the specifications of that element, each with its own unique Id.

```
   create_NeuroLayers(Cx_Id,Sensor,Actuator,LayerDensities) ->
         Input_IdPs = [{Sensor#sensor.id,Sensor#sensor.vl}],
         Tot_Layers = length(LayerDensities),
         [FL_Neurons|Next_LDs] = LayerDensities,
         NIds = [{neuron,{1,Id}}|| Id <- generate_ids(FL_Neurons,[])],
         create_NeuroLayers(Cx_Id,Actuator#actuator.id,1,Tot_Layers,Input_IdPs,NIds,Next_LDs,[]).
```

%The function create_NeuroLayers/3 prepares the initial step before starting the recursive create_NeuroLayers/7 function which will create all the Neuron records. We first generate the place holder Input Ids "Plus"(Input_IdPs), which are tuples composed of Ids and the vector lengths of the incoming signals associated with them. The proper input_idps will have a weight list in the tuple instead of the vector length. Because we are only building NNs each with only a single Sensor and Actuator, the IdP to the first layer is composed of the single Sensor Id with the vector length of its sensory signal, likewise in the case of the Actuator. We then generate unique ids for the neurons in the first layer, and drop into the recursive create_NeuroLayers/7 function.

```
   create_NeuroLayers(Cx_Id,Actuator_Id,LayerIndex,Tot_Layers,Input_IdPs,NIds,[Next_LD|LDs],Acc) ->
         Output_NIds = [{neuron,{LayerIndex+1,Id}} || Id <- generate_ids(Next_LD,[])],
         Layer_Neurons = create_NeuroLayer(Cx_Id,Input_IdPs,NIds,Output_NIds,[]),
         Next_InputIdPs = [{NId,1}|| NId <- NIds],
         create_NeuroLayers(Cx_Id,Actuator_Id,LayerIndex+1,Tot_Layers,Next_InputIdPs,Output_NIds,LDs,[Layer_Neurons|Acc]);
   create_NeuroLayers(Cx_Id,Actuator_Id,Tot_Layers,Tot_Layers,Input_IdPs,NIds,[],Acc) ->
         Output_Ids = [Actuator_Id],
         Layer_Neurons = create_NeuroLayer(Cx_Id,Input_IdPs,NIds,Output_Ids,[]),
         lists:reverse([Layer_Neurons|Acc]).
```

%During the first iteration, the first layer neuron ids constructed in create_NeuroLayers/3 are held in the NIds variable. In create_NeuroLayers/7, with every iteration we generate the Output_NIds, which are the Ids of the neurons in the next layer. The last layer is a special case which occurs when LayerIndex == Tot_Layers. Having the Input_IdPs, and the Output_NIds, we are able to construct a neuron record for every Id in NIds using the function create_layer/4. The Ids of the constructed Output_NIds will become the NIds variable of the next iteration, and the Ids of the neurons in the current layer will be extended and become Next_InputIdPs. We

```
then drop into the next iteration with the newly prepared Next_InputIdPs and Output_NIds. Fi-
nally, when we reach the last layer, the Output_Ids is the list containing a single Id of the Actu-
ator element. We use the same function, create_NeuroLayer/4, to construct the last layer and re-
turn the result.

        create_NeuroLayer(Cx_Id,Input_IdPs,[Id|NIds],Output_Ids,Acc) ->
              Neuron = create_Neuron(Input_IdPs,Id,Cx_Id,Output_Ids),
              create_NeuroLayer(Cx_Id,Input_IdPs,NIds,Output_Ids,[Neuron|Acc]);
        create_NeuroLayer(_Cx_Id,_Input_IdPs,[],_Output_Ids,Acc) ->
              Acc.
%To create neurons from the same layer, all that is needed are the Ids for those neurons, a list
of Input_IdPs for every neuron so that we can create the proper number of weights, and a list of
Output_Ids. Since in our simple feed forward neural network all neurons are fully connected to
the neurons in the next layer, the Input_IdPs and Output_Ids are the same for every neuron be-
longing to the same layer.

        create_Neuron(Input_IdPs,Id,Cx_Id,Output_Ids)->
              Proper_InputIdPs = create_NeuralInput(Input_IdPs,[]),
              #neuron{id=Id,cx_id =
Cx_Id,af=tanh,input_idps=Proper_InputIdPs,output_ids=Output_Ids}.

              create_NeuralInput([{Input_Id,Input_VL}|Input_IdPs],Acc) ->
                    Weights = create_NeuralWeights(Input_VL,[]),
                    create_NeuralInput(Input_IdPs,[{Input_Id,Weights}|Acc]);
              create_NeuralInput([],Acc)->
                    lists:reverse([{bias,random:uniform()-0.5}|Acc]).

                    create_NeuralWeights(0,Acc) ->
                          Acc;
                    create_NeuralWeights(Index,Acc) ->
                          W = random:uniform()-0.5,
                          create_NeuralWeights(Index-1,[W|Acc]).
%Each neuron record is composed by the create_Neuron/3 function. The create_Neuron/3 func-
tion creates the Input list from the tuples [{Id,Weights}...] using the vector lengths specified in
the place holder Input_IdPs. The create_NeuralInput/2 function uses create_NeuralWeights/2 to
generate the random weights in the range of -0.5 to 0.5, adding the bias to the end of the list.

              generate_ids(0,Acc) ->
                    Acc;
              generate_ids(Index,Acc)->
                    Id = generate_id(),
                    generate_ids(Index-1,[Id|Acc]).

              generate_id() ->
                    {MegaSeconds,Seconds,MicroSeconds} = now(),
                        1/(MegaSeconds*1000000 + Seconds + MicroSeconds/1000000).
%The generate_id/0 creates a unique Id using current time, the Id is a floating point value. The
generate_ids/2 function creates a list of unique Ids.

   create_Cortex(Cx_Id,S_Ids,A_Ids,NIds) ->
          #cortex{id = Cx_Id, sensor_ids=S_Ids, actuator_ids=A_Ids, nids = NIds}.
%The create_Cortex/4 function generates the record encoded genotypical representation of the
cortex element. The Cortex element needs to know the Id of every Neuron, Sensor, and Actua-
tor in the NN.
```

Note that the constructor can only create sensor and actuator records that are specified in the create_Sensor/1 and create_Actuator/1 functions, and it can only create the genotype if it knows the Sensor and Actuator **vl** parameters. Let us now compile and test our genotype constructing algorithm:

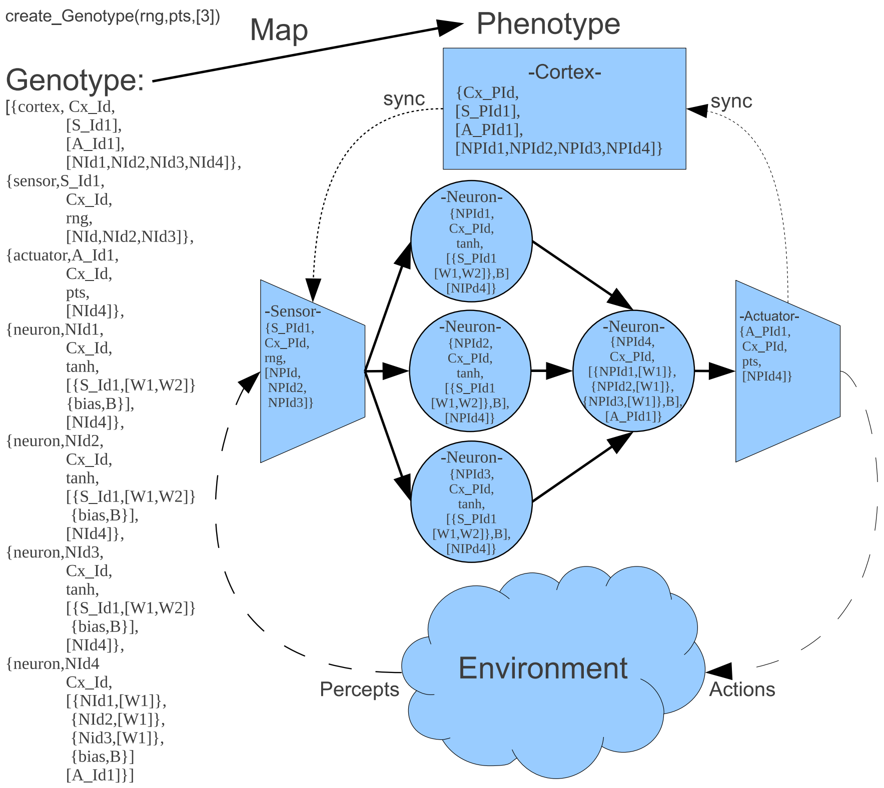
```
1>c(constructor).
{ok,constructor}.
2>constructor:construct_Genotype(ffnn,rng,pts,[1,3]).
ok
```

It works! Make sure to open the file to which the Genotype was written (ffnn in the above example), and peruse the generated list of tuples to ensure that all the elements are properly interconnected by looking at their fanin/fanout and input/output ids. This list is a genotype of the NN which is composed of 3 feed forward neural layers, with 1 neuron in the first layer, 3 in the second, and 1 in the third. The created NN genotype uses the **rng** sensor and **pts** actuator. In the next section we will create a genotype to phenotype mapper which will convert inert genotypes of this form, into live phenotypes which can process sensory signals and act on the world using their actuators.

6.6 Developing the Genotype to Phenotype Mapping Module

We've invented a tuple based genotype representation for our Neural Network, and we have developed an algorithm which creates the NN genotypes when provided with 3 high level parameters, SensorName, ActuatorName, and HiddenLayerDensities. But our genotypical representation of the NN is only used as a method of storing it in a database or some file. We now need to create a function that converts the NN genotype, to an active phenotype.

In the previous chapter we have discussed how Erlang, unlike any other language, is perfect for developing fault tolerant and concurrent NN systems. The NN topology and functionality maps perfectly to Erlang's process based architecture. We now need to design an algorithm that creates a process for every tuple encoded element (Cortex, Neurons, Actuator, Sensor) stored in the genotype, and then interconnects those processes to produce the proper NN topology. This mapping is an example of direct encoding, where every tuple becomes a process, and every connection is explicitly specified in the genotype. The mapping is shown in Figure-6.10.



```
[{cortex, Cx_Id,
          [S_Id1],
          [A_Id1],
          [NId1,NId2,NId3,NId4]},
{sensor,S_Id1,
          Cx_Id,
          rng,
          [NId,NId2,NId3]},
{actuator,A_Id1,
          Cx_Id,
          pts,
          [NId4]},
{neuron,NId1,
          Cx_Id,
          tanh,
          [{S_Id1,[W1,W2]}
          {bias,B}],
          [NId4]},
{neuron,NId2,
          Cx_Id,
          tanh,
          [{S_Id1,[W1,W2]}
           {bias,B}],
          [NId4]},
{neuron,NId3,
          Cx_Id,
          tanh,
          [{S_Id1,[W1,W2]}
           {bias,B}],
          [NId4]},
{neuron,NId4
          Cx_Id,
          [{NId1,[W1]},
           {NId2,[W1]},
           {Nid3,[W1]},
           {bias,B}]
          [A_Id1]}]
```

Fig. 6.10 A direct genotype to phenotype mapping.

In our genotype to phenotype direct mapping, we first spawn every element to create a correlation from Ids to their respective process PIds, and then initialize every process's state using the information in its correlated record. But to get these processes to communicate, we still need to standardize the messages they will exchange between each other.

Because we want our neurons to be ambivalent to whether the signal is coming from another neuron or a sensor, all signal vector messages must be of the same form. We can let the messages passed from sensors and neurons to other neurons and actuators use the following form: {Sender_PId, forward, Signal_Vector}. The Sender_PId will allow the Neurons to match the Signal_Vector with its appropriate weight vector.

Once the actuator has accumulated all the incoming neural signals, it should be able to notify the cortex element of this, so that the cortex can trigger the sensor processes to poll for new sensory data. The actuators will use the following messages for this task: {Actuator_PId,sync}. Once the cortex has received the sync messages from all the actuators connected to its NN, it will trigger all the sensors using a messages of the form: {Cx_PId,sync}. Finally, every element other than the cortex will also accept a message of the form: {Cx_PId,terminate}. The cortex itself should be able to receive the simple 'terminate' message. In this manner we can request that the cortex terminates all the elements in the NN it oversees, and then terminates itself.

Now that we know what messages the processes will be exchanging, and how the phenotype is represented, we can start developing the cortex, sensor, actuator, neuron, and the phenotype constructor module we'll call exoself. The 'exoself' module will not only contain the algorithm that maps the genotype to phenotype, but also a function that maps the phenotype back to genotype. The phenotype to genotype mapping is a backup procedure, which will allow us to backup phenotypes that have learned something new, back to the database.

We now create the cortex module:

```
cortex.erl
-module(cortex).
-compile(export_all).
-include("records.hrl").

gen(ExoSelf_PId,Node)->
   spawn(Node,?MODULE,loop,[ExoSelf_PId]).

loop(ExoSelf_PId) ->
   receive
         {ExoSelf_PId,{Id,SPIds,APIds,NPIds},TotSteps} ->
               [SPId ! {self(),sync} || SPId <- SPIds],
               loop(Id,ExoSelf_PId,SPIds,{APIds,APIds},NPIds,TotSteps)
   end.
%The gen/2 function spawns the cortex element, which immediately starts to wait for a the
state message from the same process that spawned it, exoself. The initial state message contains
the sensor, actuator, and neuron PId lists. The message also specifies how many total Sense-
Think-Act cycles the Cortex should execute before terminating the NN system. Once we im-
plement the learning algorithm, the termination criteria will depend on the fitness of the NN, or
some other useful property

loop(Id,ExoSelf_PId,SPIds,{_APIds,MAPIds},NPIds,0) ->
   io:format("Cortex:~p is backing up and terminating.~n",[Id]),
   Neuron_IdsNWeights = get_backup(NPIds,[]),
   ExoSelf_PId ! {self(),backup,Neuron_IdsNWeights},
    [PId ! {self(),terminate} || PId <- SPIds],
    [PId ! {self(),terminate} || PId <- MAPIds],
    [PId ! {self(),termiante} || PId <- NPIds];
loop(Id,ExoSelf_PId,SPIds,{[APId|APIds],MAPIds},NPIds,Step) ->
    receive
            {APId,sync} ->
                    loop(Id,ExoSelf_PId,SPIds,{APIds,MAPIds},NPIds,Step);
            terminate ->
                    io:format("Cortex:~p is terminating.~n",[Id]),
                    [PId ! {self(),terminate} || PId <- SPIds],
                    [PId ! {self(),terminate} || PId <- MAPIds],
                    [PId ! {self(),termiante} || PId <- NPIds]
    end;
loop(Id,ExoSelf_PId,SPIds,{[],MAPIds},NPIds,Step)->
    [PId ! {self(),sync} || PId <- SPIds],
    loop(Id,ExoSelf_PId,SPIds,{MAPIds,MAPIds},NPIds,Step-1).
%The cortex's goal is to synchronize the NN system such that when the actuators have received
all their control signals, the sensors are once again triggered to gather new sensory information.
Thus the cortex waits for the sync messages from the actuator PIds in its system, and once it has
received all the sync messages, it triggers the sensors and then drops back to waiting for a new
set of sync messages. The cortex stores 2 copies of the actuator PIds: the APIds, and the
MemoryAPIds (MAPIds). Once all the actuators have sent it the sync messages, it can restore
the APIds list from the MAPIds. Finally, there is also the Step variable which decrements every
time a full cycle of Sense-Think-Act completes, once this reaches 0, the NN system begins its
termination and backup process.

    get_backup([NPId|NPIds],Acc)->
            NPId ! {self(),get_backup},
            receive
                    {NPId,NId,WeightTuples}->
                            get_backup(NPIds,[{NId,WeightTuples}|Acc])
            end;
    get_backup([],Acc)->
            Acc.
%During backup, cortex contacts all the neurons in its NN and requests for the neuron's Ids and
their Input_IdPs. Once the updated Input_IdPs from all the neurons have been accumulated, the
list is sent to exoself for the actual backup and storage.
```

Now the sensor module:

```
sensor.erl
-module(sensor).
-compile(export_all).
-include("records.hrl").
```

```
gen(ExoSelf_PId,Node)->
    spawn(Node,?MODULE,loop,[ExoSelf_PId]).

loop(ExoSelf_PId) ->
    receive
            {ExoSelf_PId,{Id,Cx_PId,SensorName,VL,Fanout_PIds}} ->
                    loop(Id,Cx_PId,SensorName,VL,Fanout_PIds)
    end.
%When gen/2 is executed it spawns the sensor element and immediately begins to wait for its
initial state message.

loop(Id,Cx_PId,SensorName,VL,Fanout_PIds)->
    receive
            {Cx_PId,sync}->
                    SensoryVector = sensor:SensorName(VL),
                    [Pid ! {self(),forward,SensoryVector} || Pid <- Fanout_PIds],
                    loop(Id,Cx_PId,SensorName,VL,Fanout_PIds);
            {Cx_PId,terminate} ->
                    ok
    end.
%The sensor process accepts only 2 types of messages, both from the cortex. The sensor can ei-
ther be triggered to begin gathering sensory data based on its sensory role, or terminate if the
cortex requests so.

rng(VL)->
    rng(VL,[]).
rng(0,Acc)->
    Acc;
rng(VL,Acc)->
    rng(VL-1,[random:uniform()|Acc]).

%'rng' is a simple random number generator that produces a vector of random values, each be-
tween 0 and 1. The length of the vector is defined by the VL, which itself is specified within the
sensor record.
```

The actuator module:

```
actuator.erl
-module(actuator).
-compile(export_all).
-include("records.hrl").

gen(ExoSelf_PId,Node)->
    spawn(Node,?MODULE,loop,[ExoSelf_PId]).
```

```
loop(ExoSelf_PId) ->
    receive
            {ExoSelf_PId,{Id,Cx_PId,ActuatorName,Fanin_PIds}} ->
                    loop(Id,Cx_PId,ActuatorName,{Fanin_PIds,Fanin_PIds},[])
    end.
%When gen/2 is executed it spawns the actuator element and immediately begins to wait for its
initial state message.

loop(Id,Cx_PId,AName,{[From_PId|Fanin_PIds],MFanin_PIds},Acc) ->
    receive
            {From_PId,forward,Input} ->
                    loop(Id,Cx_PId,AName,{Fanin_PIds,MFanin_PIds},lists:append(Input,Acc));
            {Cx_PId,terminate} ->
                    ok
    end;
loop(Id,Cx_PId,AName,{[],MFanin_PIds},Acc)->
    actuator:AName(lists:reverse(Acc)),
    Cx_PId ! {self(),sync},
    loop(Id,Cx_PId,AName,{MFanin_PIds,MFanin_PIds},[]).
%The actuator process gathers the control signals from the neurons, appending them to the ac-
cumulator. The order in which the signals are accumulated into a vector is in the same order as
the neuron ids are stored within NIds. Once all the signals have been gathered, the actuator
sends cortex the sync signal, executes its function, and then again begins to wait for the neural
signals from the output layer by reseting the Fanin_PIds from the second copy of the list.

pts(Result)->
    io:format("actuator:pts(Result): ~p~n",[Result]).
%The pts actuation function simply prints to screen the vector passed to it.
```

And finally the neuron module:

```
neuron.erl
-module(neuron).
-compile(export_all).
-include("records.hrl").

gen(ExoSelf_PId,Node)->
    spawn(Node,?MODULE,loop,[ExoSelf_PId]).

loop(ExoSelf_PId) ->
    receive
            {ExoSelf_PId,{Id,Cx_PId,AF,Input_PIdPs,Output_PIds}} ->
                    loop(Id,Cx_PId,AF,{Input_PIdPs,Input_PIdPs},Output_PIds,0)
    end.
```

```
%When gen/2 is executed it spawns the neuron element and immediately begins to wait for its
initial state message.

loop(Id,Cx_PId,AF,{[{Input_PId,Weights}|Input_PIdPs],MInput_PIdPs},Output_PIds,Acc)->
    receive
            {Input_PId,forward,Input}->
                    Result = dot(Input,Weights,0),
                    loop(Id,Cx_PId,AF,{Input_PIdPs,MInput_PIdPs},Output_PIds,Result+Acc);
            {Cx_PId,get_backup}->
                    Cx_PId ! {self(),Id,MInput_PIdPs},

    loop(Id,Cx_PId,AF,{[{Input_PId,Weights}|Input_PIdPs],MInput_PIdPs},Output_PIds,Acc);
            {Cx_PId,terminate}->
                    ok
    end;
loop(Id,Cx_PId,AF,{[Bias],MInput_PIdPs},Output_PIds,Acc)->
    Output = neuron:AF(Acc+Bias),
    [Output_PId ! {self(),forward,[Output]} || Output_PId <- Output_PIds],
    loop(Id,Cx_PId,AF,{MInput_PIdPs,MInput_PIdPs},Output_PIds,0);
loop(Id,Cx_PId,AF,{[],MInput_PIdPs},Output_PIds,Acc)->
    Output = neuron:AF(Acc),
    [Output_PId ! {self(),forward,[Output]} || Output_PId <- Output_PIds],
    loop(Id,Cx_PId,AF,{MInput_PIdPs,MInput_PIdPs},Output_PIds,0).

    dot([I|Input],[W|Weights],Acc) ->
            dot(Input,Weights,I*W+Acc);
    dot([],[],Acc)->
            Acc.
%The neuron process waits for vector signals from all the processes that it's connected from,
taking the dot product of the input and weight vectors, and then adding it to the accumulator.
Once all the signals from Input_PIds are received, the accumulator contains the dot product to
which the neuron then adds the bias and executes the activation function on. After fanning out
the output signal, the neuron again returns to waiting for incoming signals. When the neuron re-
ceives the {Cx_PId,get_backup} message, it forwards to the cortex its full MInput_PIdPs list,
and its Id. Once the training/learning algorithm is added to the system, the MInput_PIdPs
would contain a full set of the most recent and updated version of the weights.

    tanh(Val)->
            math:tanh(Val).
%Though in this current implementation the neuron has only the tanh/1 function available to it,
we will later extend the system to allow different neurons to use different activation functions.
```

Now we create the exoself module, which will map the genotype to phenotype, spawning all the appropriate processes. The exoself module will also provide the algorithm for the Cortex element to update the genotype with the newly trained weights from the phenotype, and in this manner saving the trained and learned NNs for future use.

```
exoself.erl
-module(exoself).
-compile(export_all).
-include("records.hrl").

map()->
    map(ffnn).
map(FileName)->
    {ok,Genotype} = file:consult(FileName),
    spawn(exoself,map,[FileName,Genotype]).
map(FileName,Genotype)->
    IdsNPIds = ets:new(idsNpids,[set,private]),
    [Cx|CerebralUnits] = Genotype,
    Sensor_Ids = Cx#cortex.sensor_ids,
    Actuator_Ids = Cx#cortex.actuator_ids,
    NIds = Cx#cortex.nids,
    spawn_CerebralUnits(IdsNPIds,cortex,[Cx#cortex.id]),
    spawn_CerebralUnits(IdsNPIds,sensor,Sensor_Ids),
    spawn_CerebralUnits(IdsNPIds,actuator,Actuator_Ids),
    spawn_CerebralUnits(IdsNPIds,neuron,NIds),
    link_CerebralUnits(CerebralUnits,IdsNPIds),
    link_Cortex(Cx,IdsNPIds),
    Cx_PId = ets:lookup_element(IdsNPIds,Cx#cortex.id,2),
    receive
            {Cx_PId,backup,Neuron_IdsNWeights}->
                    U_Genotype = update_genotype(IdsNPIds,Genotype,Neuron_IdsNWeights),
                    {ok, File} = file:open(FileName, write),
                    lists:foreach(fun(X) -> io:format(File, "~p.~n",[X]) end, U_Genotype),
                    file:close(File),
                    io:format("Finished updating to file:~p~n",[FileName])
    end.
%The map/1 function maps the tuple encoded genotype into a process based phenotype. The
map function expects for the Cx record to be the leading tuple in the tuple list it reads from the
FileName. We create an ets table to map Ids to PIds and back again. Since the Cortex element
contains all the Sensor, Actuator, and Neuron Ids, we are able to spawn each neuron using its
own gen function, and in the process construct a map from Ids to PIds. We then use
link_CerebralUnits to link all non Cortex elements to each other by sending each spawned pro-
cess the information contained in its record, but with Ids converted to Pids where appropriate.
Finally, we provide the Cortex process with all the PIds in the NN system by executing the
link_Cortex/2 function. Once the NN is up and running, exoself starts its wait until the NN has
finished its job and is ready to backup. When the cortex initiates the backup process it sends
exoself the updated Input_PIdPs from its neurons. Exoself uses the update_genotype/3 function
```

```
to update the old genotype with new weights, and then stores the updated version back to its
file.

   spawn_CerebralUnits(IdsNPIds,CerebralUnitType,[Id|Ids])->
         PId = CerebralUnitType:gen(self(),node()),
         ets:insert(IdsNPIds,{Id,PId}),
         ets:insert(IdsNPIds,{PId,Id}),
         spawn_CerebralUnits(IdsNPIds,CerebralUnitType,Ids);
   spawn_CerebralUnits(_IdsNPIds,_CerebralUnitType,[])->
         true.
%We spawn the process for each element based on its type: CerebralUnitType, and the gen
function that belongs to the CerebralUnitType module. We then enter the {Id,PId} tuple into
our ETS table for later use.

   link_CerebralUnits([R|Records],IdsNPIds) when is_record(R,sensor) ->
         SId = R#sensor.id,
         SPId = ets:lookup_element(IdsNPIds,SId,2),
         Cx_PId = ets:lookup_element(IdsNPIds,R#sensor.cx_id,2),
         SName = R#sensor.name,
         Fanout_Ids = R#sensor.fanout_ids,
         Fanout_PIds = [ets:lookup_element(IdsNPIds,Id,2) || Id <- Fanout_Ids],
         SPId ! {self(),{SId,Cx_PId,SName,R#sensor.vl,Fanout_PIds}},
         link_CerebralUnits(Records,IdsNPIds);
   link_CerebralUnits([R|Records],IdsNPIds) when is_record(R,actuator) ->
         AId = R#actuator.id,
         APId = ets:lookup_element(IdsNPIds,AId,2),
         Cx_PId = ets:lookup_element(IdsNPIds,R#actuator.cx_id,2),
         AName = R#actuator.name,
         Fanin_Ids = R#actuator.fanin_ids,
         Fanin_PIds = [ets:lookup_element(IdsNPIds,Id,2) || Id <- Fanin_Ids],
         APId ! {self(),{AId,Cx_PId,AName,Fanin_PIds}},
         link_CerebralUnits(Records,IdsNPIds);
   link_CerebralUnits([R|Records],IdsNPIds) when is_record(R,neuron) ->
         NId = R#neuron.id,
         NPId = ets:lookup_element(IdsNPIds,NId,2),
         Cx_PId = ets:lookup_element(IdsNPIds,R#neuron.cx_id,2),
         AFName = R#neuron.af,
         Input_IdPs = R#neuron.input_idps,
         Output_Ids = R#neuron.output_ids,
         Input_PIdPs = convert_IdPs2PIdPs(IdsNPIds,Input_IdPs,[]),
         Output_PIds = [ets:lookup_element(IdsNPIds,Id,2) || Id <- Output_Ids],
         NPId ! {self(),{NId,Cx_PId,AFName,Input_PIdPs,Output_PIds}},
         link_CerebralUnits(Records,IdsNPIds);
   link_CerebralUnits([],_IdsNPIds)->
         ok.
```

```
        convert_IdPs2PIdPs(_IdsNPIds,[{bias,Bias}],Acc)->
            lists:reverse([Bias|Acc]);
        convert_IdPs2PIdPs(IdsNPIds,[{Id,Weights}|Fanin_IdPs],Acc)->
            convert_IdPs2PIdPs(IdsNPIds,Fanin_IdPs,
[{ets:lookup_element(IdsNPIds,Id,2),Weights}|Acc]).
%The link_CerebralUnits/2 converts the Ids to PIds using the created IdsNPids ETS table. At
this point all the elements are spawned, and the processes are waiting for their initial states.
convert_IdPs2PIdPs/3 converts the IdPs tuples into tuples that use PIds instead of Ids, such that
the Neuron will know which weights are to be associated with which incoming vector signals.
The last element is the bias, which is added to the list in a non tuple form. Afterwards, the list is
reversed to take its proper order.

    link_Cortex(Cx,IdsNPIds) ->
        Cx_Id = Cx#cortex.id,
        Cx_PId = ets:lookup_element(IdsNPIds,Cx_Id,2),
        SIds = Cx#cortex.sensor_ids,
        AIds = Cx#cortex.actuator_ids,
        NIds = Cx#cortex.nids,
        SPIds = [ets:lookup_element(IdsNPIds,SId,2) || SId <- SIds],
        APIds = [ets:lookup_element(IdsNPIds,AId,2) || AId <- AIds],
        NPIds = [ets:lookup_element(IdsNPIds,NId,2) || NId <- NIds],
        Cx_PId ! {self(),{Cx_Id,SPIds,APIds,NPIds},1000}.
%The cortex is initialized to its proper state just as other elements. Because we have not yet
implemented a learning algorithm for our NN system, we need to specify when the NN should
shutdown. We do this by specifying the total number of cycles the NN should execute before
terminating, which is 1000 in this case.

update_genotype(IdsNPIds,Genotype,[{N_Id,PIdPs}|WeightPs])->
    N = lists:keyfind(N_Id, 2, Genotype),
    io:format("PIdPs:~p~n",[PIdPs]),
    Updated_InputIdPs = convert_PIdPs2IdPs(IdsNPIds,PIdPs,[]),
    U_N = N#neuron{input_idps = Updated_InputIdPs},
    U_Genotype = lists:keyreplace(N_Id, 2, Genotype, U_N),
    io:format("N:~p~n U_N:~p~n Genotype:~p~n
U_Genotype:~p~n",[N,U_N,Genotype,U_Genotype]),
    update_genotype(IdsNPIds,U_Genotype,WeightPs);
update_genotype(_IdsNPIds,Genotype,[])->
    Genotype.

    convert_PIdPs2IdPs(IdsNPIds,[{PId,Weights}|Input_PIdPs],Acc)->
        con-
vert_PIdPs2IdPs(IdsNPIds,Input_PIdPs,[{ets:lookup_element(IdsNPIds,PId,2),Weights}|Acc]);
    convert_PIdPs2IdPs(_IdsNPIds,[Bias],Acc)->
        lists:reverse([{bias,Bias}|Acc]).
```

```
%For every {N_Id,PIdPs} tuple the update_genotype/3 function extracts the neuron with the id:
N_Id, and updates its weights. The convert_PIdPs2IdPs/3 performs the conversion from PIds to
Ids of every {PId,Weights} tuple in the Input_PIdPs list. The updated Genotype is then returned
back to the caller.
```

Now lets compile the cortex, neuron, sensor, actuator, and the exoself module, and test the NN system:

```
1> c(cortex).
ok
…
```

We now create a new NN genotype which uses the rng sensor, a pts actuator, and employs a [1,2] hidden density list. Then we map it to its phenotype by using the exoself module.

```
1> constructor:construct_Genotype(ffnn,rng,pts,[1,2]).
ok
2> exoself:map(ffnn).
...
```

It works! Our NN system has sensed, thought, and acted using its sensor, neurons, and the actuator, while being synchronized through the cortex process. Yet still this system does nothing but process random vectors using neural processes which themselves use random weights. We now need to develop a learning algorithm, and then devise a problem on which to test how well the NN can learn and solve problems using its learning method. In the next chapter we will develop an augmented version one of the most commonly used unsupervised learning algorithms: the Stochastic Hill-Climber.

6.7 Summary

We have started with just a discussion of how a single artificial neuron processes an incoming signal, which is vector encoded. We then developed a simple sensor and actuator, so that the neuron has something to acquire sensory signals with, and so that it can use its output signal to act upon the world, in this case simply printing that output signal to screen. We then began designing the architecture of the NN system we wish to develop, and the genotype encoding we wanted to store that NN system in. After we had agreed on the architecture and the encoding, we created the genotype constructor which built the NN genotype, and then a mapper function which converted the genotype to its phenotype form. With this, we had now developed a system that can create NN genotypes, and convert them to phenotypes, We tested the ability of the NN to sense using its sensors, thinking about the signals it acquired through its sensors, and then act upon the world by using its actuators; the system worked. Though our NN system does not yet have a way to learn, or be optimized for any particular task, we have developed a complete encoding method, a genotypical and phenotypical representation of a fully concurrent NN system, in just a few pages. With Erlang, a neuron is a process, an action potential is a message, there is a 1:1 mapping, which made developing this system so easy.