Chapter 9 Testing the Neuroevolutionary System

Abstract In this chapter we test the newly created basic neuroevolutionary system, by first testing each of its mutation operators, and then by applying the whole system to the XOR mimicking problem. Though the XOR problem test will run to completion and without errors, a more detailed, manual analysis of the evolved topologies and genotypes of the fit agents will show a number of bugs to be present. The origins of the bugs is then analyzed, and the errors are fixed. Afterwards, the updated neuroevolutionary system is then successfully re-tested.

9.1 Testing the Mutation Operators

Having created the basic neuroevolutionary system, we need to test whether the mutation operators work as we intended them to. We have set up all the complexifying mutation operators to leave the system in a connected state. This means that when we apply these mutation operators, the resulting NN topology is such, that the signal can get from the sensors, all the way through the NN, and to the actuators. The pruning mutation operators: remove_inlink, remove_outlink, remove_neuron, remove_sensor, remove_actuator, may leave the NN in such a state that it is no longer able to process information, by creating a break in the connected graph, as shown in the example of Fig-9.1. We could start using the pruning mutation operators later on, after we have first created a program inside the genome_mutator module that ensures that all the resulting mutant NN systems are not disconnected after such pruning mutation operators have been applied.

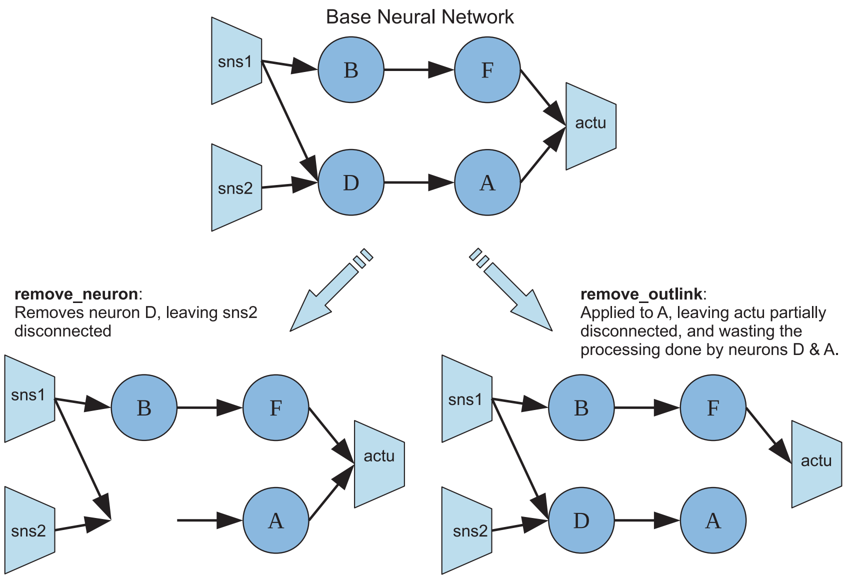

Base Neural Network
sns1
sns2
B
F
D
A
actu
remove_neuron:
Removes neuron D, leaving sns2 disconnected
remove_outlink:
Applied to A, leaving actu partially disconnected, and wasting the processing done by neurons D & A.

Fig. 9.1 Pruning mutation operators that leave a NN disconnected.

Let us now run a few mutation operator tests, to see if the resulting topologies after we have applied some mutation operators to the NN, are as expected. When you perform the same tests, the results may slightly differ from mine, since the elements in your NN will have different Ids, and because the mutation operators are applied randomly. The test of each mutation operator will have the following steps:

1. Generate a *test* NN, which is composed of a single neuron, connected from the sensor *xor_GetInput*, and connected to the actuator *xor_SendOutput*. This is done by simply executing *genotype:create_test()*, which creates a *xor_mimic* morphology based seed agent.
2. Apply an available mutation operator by executing: genome_mutator:test(test, Mutator).
3. Execute genotype:print(test) to print the resulting genotype to console, and then compare it to the original genotype to ensure that the resulting mutated genotype is as expected based on the mutation operator used.
4. Test the resulting NN on the simple XOR problem for which it has the sensor and actuator, by executing exoself:start(test,void). There will not exist a population_monitor process at this time, but that should not affect the results. The goal here is to ensure that the NN does not stall, that the signals can go all the way through it, from sensors to actuators, and that the NN system is functional. In this case we do not expect the NN to solve the problem, because the topology is not evolving towards any particular goal.

Let us now go through these steps for each mutation operator. For the sake of being brief, I will show the entire console printout for the first mutation operator test, but for all the other mutation operators I will only display the most significant console printout parts.

mutate_weights: This mutation operator selects a random neuron in the NN and perturbs/mutates its synaptic weights.

```
2> genotype:create_test().
{agent,test,0,undefined,test,
    {{origin,7.572689688224582e-10},cortex},
    {[{0,1}],
    [],
    [{sensor,undefined,xor_GetInput,undefined,
         {private,xor_sim},
         2,
         [{{0,7.572689688218573e-10},neuron}],
         undefined}],
    [{actuator,undefined,xor_SendOutput,undefined,
          {private,xor_sim},
          1,
          [{{0,7.572689688218573e-10},neuron}],
          undefined}]},
    {constraint,xor_mimic,[tanh,cos,gauss,abs]},
    [],undefined,0,
    [{0,[{{0,7.572689688218573e-10},neuron}]}]}
{cortex,{{origin,7.572689688224582e-10},cortex},
    test,
    [{{0,7.572689688218573e-10},neuron}],
    [{{-1,7.572689688218636e-10},sensor}],
    [{{1,7.572689688218589e-10},actuator}]}
{sensor,{{-1,7.572689688218636e-10},sensor},
    xor_GetInput,
    {{origin,7.572689688224582e-10},cortex},
    {private,xor_sim},
    2,
    [{{0,7.572689688218573e-10},neuron}],
    undefined}
{neuron,{{0,7.572689688218573e-10},neuron},
    0,
    {{origin,7.572689688224582e-10},cortex},
    tanh,
    [{{{-1,7.572689688218636e-10},sensor},
      [-0.08541081650616245,-0.02882161114431025]}],
    [{{1,7.572689688218589e-10},actuator}],
    []}
{actuator,{{1,7.572689688218589e-10},actuator},
      xor_SendOutput,
      {{origin,7.572689688224582e-10},cortex},
      {private,xor_sim},
      1,
      [{{0,7.572689688218573e-10},neuron}],
      undefined}
{atomic,{atomic,[ok]}}
3> genome_mutator:test(test,mutate_weights).
{atomic,{atomic,ok}}
4> genotype:print(test).
{agent,test,0,undefined,test,
    {{origin,7.572689688224582e-10},cortex},
    {[{0,1}],
    [],
    [{sensor,undefined,xor_GetInput,undefined,
          {private,xor_sim},
          2,
          [{{0,7.572689688218573e-10},neuron}],
          undefined}],
    [{actuator,undefined,xor_SendOutput,undefined,
           {private,xor_sim},
           1,
           [{{0,7.572689688218573e-10},neuron}],
           undefined}]},
    {constraint,xor_mimic,[tanh,cos,gauss,abs]},
    [{mutate_weights,{{0,7.572689688218573e-10},neuron}}],
    undefined,0,
    [{0,[{{0,7.572689688218573e-10},neuron}]}]}
{cortex,{{origin,7.572689688224582e-10},cortex},
    test,
    [{{0,7.572689688218573e-10},neuron}],
    [{{-1,7.572689688218636e-10},sensor}],
    [{{1,7.572689688218589e-10},actuator}]}
{sensor,{{-1,7.572689688218636e-10},sensor},
    xor_GetInput,
    {{origin,7.572689688224582e-10},cortex},
    {private,xor_sim},
    2,
    [{{0,7.572689688218573e-10},neuron}],
    undefined}
{neuron,{{0,7.572689688218573e-10},neuron},
    0,
    {{origin,7.572689688224582e-10},cortex},
    tanh,
    [{{{-1,7.572689688218636e-10},sensor},
      [-1.81543903255671,0.28220989176010963]}],
    [{{1,7.572689688218589e-10},actuator}],
    []}
{actuator,{{1,7.572689688218589e-10},actuator},
      xor_SendOutput,
      {{origin,7.572689688224582e-10},cortex},
      {private,xor_sim},
      1,
      [{{0,7.572689688218573e-10},neuron}],
      undefined}
{atomic,[ok]}
```

As you can see from the printout, the mutate_weights operator chose a random neuron in the NN, which in this case is just the single existing neuron, and then mutated the synaptic weights associated with the sensor that it is connected from. The synaptic weights were mutated from their original values of:

```
[-0.08541081650616245, -0.028821611144310255]
```

to:

```
[-1.81543903255671, 0.28220989176010963].
```

We now test the mutated NN system on the problem that its morphology defines it for, the XOR mimicking problem.

```
5> exoself:start(test,void).
<0.128.0>
Finished updating genotype
Terminating the phenotype:
Cx_PId:<0.131.0>
SPIds:[<0.132.0>]
NPIds:[<0.134.0>]
APIds:[<0.133.0>]
ScapePids:[<0.130.0>]
Sensor:{{-1,7.572689688218636e-10},sensor} is terminating.
Agent:<0.128.0> terminating. Genotype has been backed up.
 Fitness:0.505631430344058
 TotEvaluations:52
 TotCycles:208
 TimeAcc:7226
Cortex:{{origin,7.572689688224582e-10},cortex} is terminating.
```

It works! The exoself ran, and after having finished tuning the weights with our augmented stochastic hill-climber algorithm, it updated the genotype, terminated the phenotype by terminating all the processes associated with it (SPIds, NPIds, APIds, and ScapePids), and then printed to screen the stats of the NN system's run: the total evaluations, total cycles, and the total time the NN system was running.

To see that the genotype was indeed updated, we can print it out again, to see what the new synaptic weights are for the single neuron of this NN system:

```
7> genotype:print(test).
...
{neuron,{{0,7.572689688218573e-10},neuron},
     0,
     {{origin,7.572689688224582e-10},cortex},
     tanh,
     [{{{-1,7.572689688218636e-10},sensor},
       [-1.81543903255671,-2.4665070928720794]}],
     [{{1,7.572689688218589e-10},actuator}],
     []}
…
```

The original synaptic weights associated with the sensor were: **[-1.81543903255671, 0.28220989176010963]** which have been tuned to the values: **[-1.81543903255671, -2.4665070928720794]**. The synaptic weight vector is of length two, and we can see that in this case only the second weight in the vector was perturbed, where as when we applied the mutation operator, it mutated only the first weight in the vector. The mutation and perturbation process is stochastic.

The system passed the test, the mutate_weights operator works, we have manually examined the resulting NN system, which has the right topology, which is the same but with a mutated synaptic weight vector. We have tested the phenotype, and have confirmed that it works. It ran for a total of 52 evaluations, so it made 52 attempts to tune the weights. We can guess that at least 50 did not work, because we know that it takes, due to the MAX_ATTEMPTS = 50 in the exoself module, 50 failing attempts before exoself gives up tuning the weights. We also know that 1 of the evaluations was the very first one, when the NN system ran with the original genotype. So we can even extrapolate that it was the second attempt, the second evaluation, during which the perturbed synaptic weights were improved in this scenario. When you perform the test, your results will most likely be different.

add_bias: This mutation operator selects a random neuron in the NN and, if the neuron's input_idps list does not already have a bias, the mutation operator adds one.

```
2> genotype:create_test().
...
{neuron,{{0,7.572678978164637e-10},neuron},
      0,
      {{origin,7.572678978164722e-10},cortex},
      gaussian,
      [{{{-1,7.572678978164681e-10},sensor},
        [0.41211176719508646,0.06709671037415732]}],
      [{{1,7.572678978164653e-10},actuator}],
      []}
...
3> genome_mutator:test(test,add_bias).
{atomic,{atomic,ok}}
4> genotype:print(test).
...
{neuron,{{0,7.572678978164637e-10},neuron},
      0,
      {{origin,7.572678978164722e-10},cortex},
      gaussian,
      [{{{-1,7.572678978164681e-10},sensor},
        [0.41211176719508646,0.06709671037415732]},
       {bias,[-0.1437300365267422]}],
      [{{1,7.572678978164653e-10},actuator}],
      []}
...
5> exoself:start(test,void).
…
```

It works! The original genotype had a neuron connected from the sensor, using a *gaussian* activation function, with the synaptic weight vector associated with the sensor: **[0.41211176719508646, 0.06709671037415732]**. After the add_bias mutation operator was executed, the neuron acquired the bias weight: **[-0.1437300365267422]**. Finally, we now test out the new NN system by converting the genotype to its phenotype by executing the exoself:start(test,void) function. As in the previous test, when I ran it with this mutated agent, there were no errors, and the system terminated normally.

mutate_af: This mutation operator selects a random neuron in the NN and changes its activation function to a new one, selected from the list available in the constraint's neural_afs list.

```
2> genotype:create_test().
...
{neuron,{{0,7.572652623199229e-10},neuron},
      0,
        {{origin,7.57265262319932e-10},cortex},
        absolute,
        [{{{-1,7.572652623199274e-10},sensor},
          [-0.16727779071660276,0.12410379914428638]}],
        [{{1,7.572652623199246e-10},actuator}],
        []}
...
3> genome_mutator:test(test,mutate_af).
{atomic,{atomic,ok}}
4> genotype:print(test).
...
{neuron,{{0,7.572652623199229e-10},neuron},
        0,
        {{origin,7.57265262319932e-10},cortex},
        cos,
        [{{{-1,7.572652623199274e-10},sensor},
          [-0.16727779071660276,0.12410379914428638]}],
        [{{1,7.572652623199246e-10},actuator}],
        []}
...
{atomic,[ok]}
25> exoself:start(test,void).
...
```

The original randomly selected activation function of the single neuron in the test agent was the *absolute* activation function. After we have applied the *mutate_af* operator to the NN system, the activation function was changed to *cos*. As before, here too converting the genotype to phenotype worked, as there were no errors when running *exoself:start(test,void)*.

add_outlink & add_inlink: The add_outlink operator chooses a random neuron and adds an output connection *from it*, to another randomly selected element in the NN system. The add_inlink operator chooses a random neuron and adds an input connection *to it*, from another randomly selected element in the NN. We will only test one of them, the add_outlink, as they both function very similarly.

```
2> genotype:create_test().
...
{sensor,{{-1,7.572648155161364e-10},sensor},
        xor_GetInput,
        {{origin,7.572648155161404e-10},cortex},
        {private,xor_sim},
        2,
        [{{0,7.572648155161313e-10},neuron}],
        undefined}
```

```
{neuron,{{0,7.572648155161313e-10},neuron},
       0,
       {{origin,7.572648155161404e-10},cortex},
       absolute,
       [{{{-1,7.572648155161364e-10},sensor},
         [-0.02132967923622686,-0.38581737041377817]}],
       [{{1,7.572648155161335e-10},actuator}],
       []}
{actuator,{{1,7.572648155161335e-10},actuator},
         xor_SendOutput,
         {{origin,7.572648155161404e-10},cortex},
         {private,xor_sim},
         1,
         [{{0,7.572648155161313e-10},neuron}],
         undefined}
{atomic,{atomic,[ok]}}
3> genome_mutator:test(test,add_outlink).
{atomic,{atomic,ok}}
4> genotype:print(test).
...
{sensor,{{-1,7.572648155161364e-10},sensor},
       xor_GetInput,
       {{origin,7.572648155161404e-10},cortex},
       {private,xor_sim},
       2,
       [{{0,7.572648155161313e-10},neuron}],
       undefined}
{neuron,{{0,7.572648155161313e-10},neuron},
       0,
       {{origin,7.572648155161404e-10},cortex},
       absolute,
       [{{{0,7.572648155161313e-10},neuron},[-0.13154644819577532]},
        {{{-1,7.572648155161364e-10},sensor},
         [-0.02132967923622686,-0.38581737041377817]}],
       [{{0,7.572648155161313e-10},neuron},
        {{1,7.572648155161335e-10},actuator}],
       [{{0,7.572648155161313e-10},neuron}]}
{actuator,{{1,7.572648155161335e-10},actuator},
         xor_SendOutput,
         {{origin,7.572648155161404e-10},cortex},
         {private,xor_sim},
         1,
         [{{0,7.572648155161313e-10},neuron}],
         undefined}
{atomic,[ok]}
```

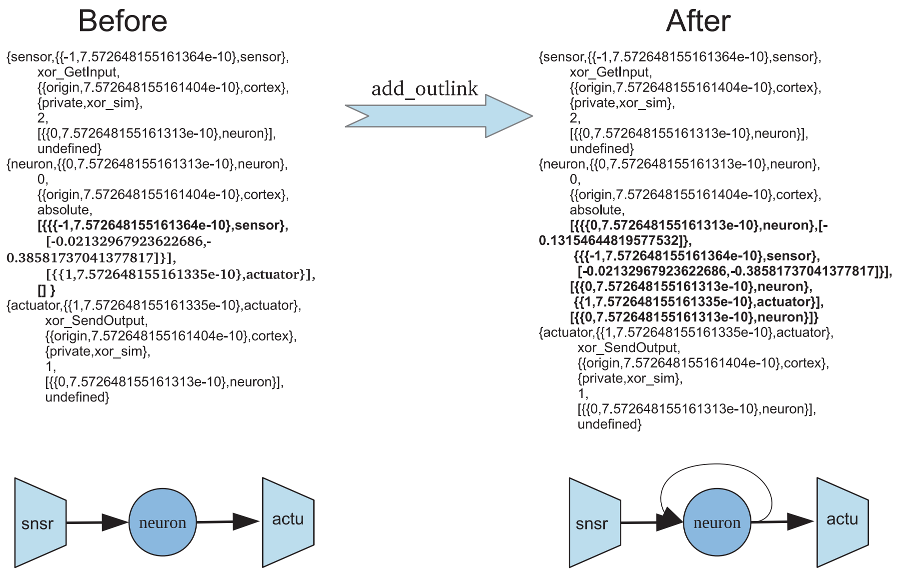
It works! The original neuron had the form:

```
{neuron,{{0,7.572648155161313e-10},neuron},
    0,
    {{origin,7.572648155161404e-10},cortex},
    absolute,
    [{{{-1,7.572648155161364e-10},sensor},
     [-0.02132967923622686,-0.3858173704137781 7]}],
    [{{1,7.572648155161335e-10},actuator}],
    []}
```

It only had a single input connection which was from the sensor, and a single output connection to the actuator. After the add_outlink operator was executed, the new NN system's neuron had the following form:

```
{neuron,{{0,7.572648155161313e-10},neuron},
    0,
    {{origin,7.572648155161404e-10},cortex},
    absolute,
    [{{{0,7.572648155161313e-10},neuron},[-0.13154644819577532]},
     {{{-1,7.572648155161364e-10},sensor},
      [-0.02132967923622686,-0.38581737041377817]}],
    [{{0,7.572648155161313e-10},neuron},
     {{1,7.572648155161335e-10},actuator}],
    [{{0,7.572648155161313e-10},neuron}]}
```

In this case the neuron formed a new synaptic connection to another randomly chosen element in the NN system, in this case that other element was itself. We can see that this new connection is recursive, and we can tell this from the last element of the neuron defining tuple, which specifies *ro_ids*, a list of recurrent link ids. There is also a new synaptic weight associated with this recurrent self connection: **{{{0,7.572648155161313e-10},neuron},[-0.13154644819577532]}**. The diagram of this NN topology before and after the mutation operator was applied, is shown in Fig-9.2.



Before

```
{sensor,{{-1,7.572648155161364e-10},sensor},
       xor_GetInput,
       {{origin,7.572648155161404e-10},cortex},
       {private,xor_sim},
       2,
       [{{0,7.572648155161313e-10},neuron}],
       undefined}
{neuron,{{0,7.572648155161313e-10},neuron},
       0,
       {{origin,7.572648155161404e-10},cortex},
       absolute,
       [{{{-1,7.572648155161364e-10},sensor},
         [-0.02132967923622686,-
0.38581737041377817]}],
         [{{1,7.572648155161335e-10},actuator}],
       [] }
{actuator,{{1,7.572648155161335e-10},actuator},
         xor_SendOutput,
         {{origin,7.572648155161404e-10},cortex},
         {private,xor_sim},
         1,
         [{{0,7.572648155161313e-10},neuron}],
         undefined}
```

After

```
{sensor,{{-1,7.572648155161364e-10},sensor},
       xor_GetInput,
       {{origin,7.572648155161404e-10},cortex},
       {private,xor_sim},
       2,
       [{{0,7.572648155161313e-10},neuron}],
       undefined}
{neuron,{{0,7.572648155161313e-10},neuron},
       0,
       {{origin,7.572648155161404e-10},cortex},
       absolute,
       [{{{0,7.572648155161313e-10},neuron},[-
0.13154644819577532]},
        {{{-1,7.572648155161364e-10},sensor},
         [-0.02132967923622686,-0.38581737041377817]}],
       [{{0,7.572648155161313e-10},neuron},
        {{1,7.572648155161335e-10},actuator}],
       [{{0,7.572648155161313e-10},neuron}]}
{actuator,{{1,7.572648155161335e-10},actuator},
         xor_SendOutput,
         {{origin,7.572648155161404e-10},cortex},
         {private,xor_sim},
         1,
         [{{0,7.572648155161313e-10},neuron}],
         undefined}
```

Fig. 9.2 The NN system topology before and after add_outlink mutation operator was applied.

We now map the genotype to phenotype, to see if the new NN system is functional:

```
5> exoself:start(test,void).
<0.101.0>
Finished updating genotype
Terminating the phenotype:
…
```

It works! Though I did not show the complete printout (which looked very similar to the first fully shown console printout), the NN system worked and terminated successfully. With this test complete, we now move to a more complex mutation operator, the addition of a new random neuron to the existing NN system.

add_neuron: This mutation operator chooses a random neural layer in the NN, and then creates a new neuron and connects it from and to, two randomly selected elements in the NN system respectively.

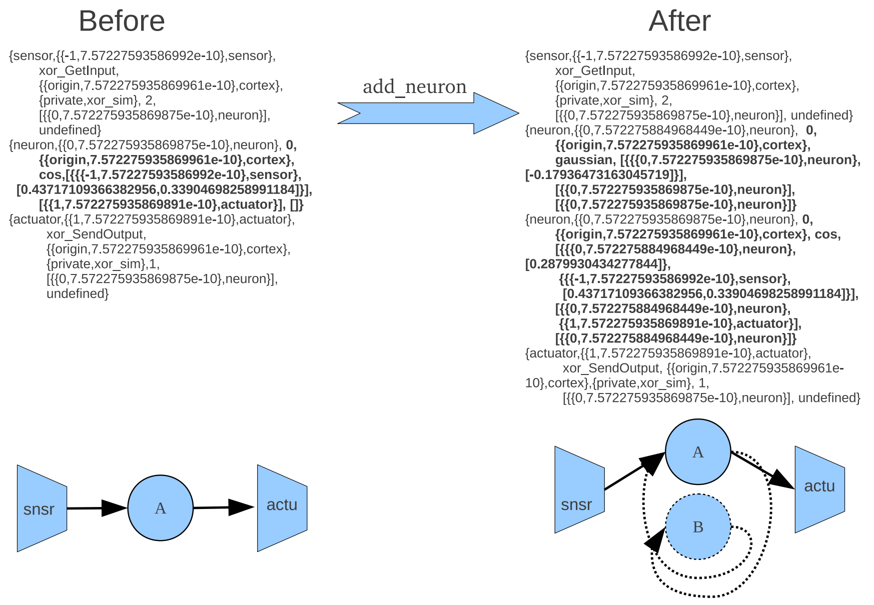
```
2> genotype:create_test().
...
{cortex,{{origin,7.572275935869961e-10},cortex},
       test,
       [{{0,7.572275935869875e-10},neuron}],
       [{{-1,7.57227593586992e-10},sensor}],
       [{{1,7.572275935869891e-10},actuator}]}
{sensor,{{-1,7.57227593586992e-10},sensor},
       xor_GetInput,
       {{origin,7.572275935869961e-10},cortex},
       {private,xor_sim},
       2,
       [{{0,7.572275935869875e-10},neuron}],
       undefined}
{neuron,{{0,7.572275935869875e-10},neuron},
       0,
       {{origin,7.572275935869961e-10},cortex},
       cos,
       [{{{-1,7.57227593586992e-10},sensor},
         [0.43717109366382956,0.33904698258991184]}],
       [{{1,7.572275935869891e-10},actuator}],
       []}
{actuator,{{1,7.572275935869891e-10},actuator},
         xor_SendOutput,
         {{origin,7.572275935869961e-10},cortex},
         {private,xor_sim},
         1,
         [{{0,7.572275935869875e-10},neuron}],
         undefined}
{atomic,{atomic,[ok]}}
3> genome_mutator:test(test,add_neuron).
{aborted,"******** ERROR:link_FromNeuronToActuator:: Actuator already fully con-
nected"}
4> genome_mutator:test(test,add_neuron).
{atomic,{atomic,ok}}
5> genotype:print(test).
...
{cortex,{{origin,7.572275935869961e-10},cortex},
       test,
       [{{0,7.572275884968449e-10},neuron},
        {{0,7.572275935869875e-10},neuron}],
       [{{-1,7.57227593586992e-10},sensor}],
       [{{1,7.572275935869891e-10},actuator}]}
{sensor,{{-1,7.57227593586992e-10},sensor},
       xor_GetInput,
       {{origin,7.572275935869961e-10},cortex},
       {private,xor_sim},
       2,
       [{{0,7.572275935869875e-10},neuron}],
       undefined}
{neuron,{{0,7.572275884968449e-10},neuron},
       0,
       {{origin,7.572275935869961e-10},cortex},
       gaussian,
       [{{{0,7.572275935869875e-10},neuron},[-0.17936473163045719]}],
       [{{0,7.572275935869875e-10},neuron}],
       [{{0,7.572275935869875e-10},neuron}]}
{neuron,{{0,7.572275935869875e-10},neuron},
       0,
       {{origin,7.572275935869961e-10},cortex},
       cos,
       [{{{0,7.572275884968449e-10},neuron},[0.2879930434277844]},
        {{{-1,7.57227593586992e-10},sensor},
         [0.43717109366382956,0.33904698258991184]}],
       [{{0,7.572275884968449e-10},neuron},
        {{1,7.572275935869891e-10},actuator}],
       [{{0,7.572275884968449e-10},neuron}]}
{actuator,{{1,7.572275935869891e-10},actuator},
         xor_SendOutput,
         {{origin,7.572275935869961e-10},cortex},
         {private,xor_sim},
         1,
         [{{0,7.572275935869875e-10},neuron}],
         undefined}
{atomic,[ok]}
```

Something very interesting happened in this test. In "**2>**" we create a new test NN system. A new NN system is fully connected to its sensors and actuators. When we try to apply the add_neuron mutation operator in "**3>**", the mutation operator must have randomly chosen to connect the new neuron to the existing actuator. But the actuator already has all the connections it needs, the vector signal it uses to execute its functionality, already has all the elements and is already connected to all the neurons it requires to function, which in this case is just a single neuron. So the mutation is rejected, as seen by the line: **{aborted,"******** ERROR:link_FromNeuronToActuator:: Actuator already fully connected"}**. During the process of neuroevolution, at this point our topology and weight evolving artificial neural network (TWEANN) system would simply try another mutation operator. Which is what I did manually in this test in "**4>**".

The new mutation worked, it created a new neuron and connected it from and to, the already existing neuron in the NN system. We can see the newly formed connection in the genotype here:

```
{neuron,{{0,7.572275884968449e-10},neuron},
     0,
     {{origin,7.572275935869961e-10},cortex},
     gaussian,
     [{{{0,7.572275935869875e-10},neuron},[-0.17936473163045719]}],
     [{{0,7.572275935869875e-10},neuron}],
     [{{0,7.572275935869875e-10},neuron}]}
{neuron,{{0,7.572275935869875e-10},neuron},
     0,
     {{origin,7.572275935869961e-10},cortex},
     cos,
     [{{{0,7.572275884968449e-10},neuron},[0.2879930434277844]},
      {{{-1,7.57227593586992e-10},sensor},
       [0.43717109366382956,0.33904698258991184]}],
     [{{0,7.572275884968449e-10},neuron},
      {{1,7.572275935869891e-10},actuator}],
     [{{0,7.572275884968449e-10},neuron}]}
```

The initial test NN system had a single neuron with the id: **{{0,7.572275935869875e-10},neuron}**, The newly added neuron has the id: **{{0,7.572275884968449e-10},neuron}**. We can see that after the mutation, both neurons have recurrent connections, which in our neuron record is represented by the last list in the tuple. The original neuron's recurrent connection list ro_ids is: **[{{0,7.572275884968449e-10},neuron}]**, containing the id of the new neuron. The newly added neuron's or_ids list is: **[{{0,7.572275935869875e-10},neuron}]**, containing in it the id of the original neuron.



```
{sensor,{{-1,7.57227593586992e-10},sensor},
        xor_GetInput,
        {{origin,7.572275935869961e-10},cortex},
        {private,xor_sim}, 2,
        [{{0,7.572275935869875e-10},neuron}],
        undefined}
{neuron,{{0,7.572275935869875e-10},neuron}, 0,
        {{origin,7.572275935869961e-10},cortex},
        cos,[{{{-1,7.57227593586992e-10},sensor},
  [0.43717109366382956,0.33904698258991184]}],
        [{{1,7.572275935869891e-10},actuator}], []}
{actuator,{{1,7.572275935869891e-10},actuator},
          xor_SendOutput,
          {{origin,7.572275935869961e-10},cortex},
          {private,xor_sim},1,
          [{{0,7.572275935869875e-10},neuron}],
          undefined}
```

```
{sensor,{{-1,7.57227593586992e-10},sensor},
        xor_GetInput,
        {{origin,7.572275935869961e-10},cortex},
        {private,xor_sim}, 2,
        [{{0,7.572275935869875e-10},neuron}], undefined}
{neuron,{{0,7.572275884968449e-10},neuron},  0,
        {{origin,7.572275935869961e-10},cortex},
        gaussian, [{{{0,7.572275935869875e-10},neuron},
[-0.17936473163045719]}],
        [{{0,7.572275935869875e-10},neuron}],
        [{{0,7.572275935869875e-10},neuron}]}
{neuron,{{0,7.572275935869875e-10},neuron}, 0,
        {{origin,7.572275935869961e-10},cortex}, cos,
        [{{{0,7.572275884968449e-10},neuron},
[0.2879930434277844]},
         {{{-1,7.57227593586992e-10},sensor},
          [0.43717109366382956,0.33904698258991184]}],
        [{{0,7.572275884968449e-10},neuron},
         {{1,7.572275935869891e-10},actuator}],
        [{{0,7.572275884968449e-10},neuron}]}
{actuator,{{1,7.572275935869891e-10},actuator},
          xor_SendOutput, {{origin,7.572275935869961e-
10},cortex},{private,xor_sim}, 1,
          [{{0,7.572275935869875e-10},neuron}], undefined}
```

Fig. 9.3 The NN system topology before and after the add_neuron mutation operator was applied.

We can also see that the new neuron is using the gaussian activation function, and that both of the neurons formed new weights for their new synaptic connections. The above figure shows the NN system's topology before and after the add_neuron mutation operator is applied.

We now test the new topology live, by mapping the genotype to its phenotype:

```
6> exoself:start(test,void).
<0.866.0>
Finished updating genotype
Terminating the phenotype:
Cx_PId:<0.868.0>
SPIds:[<0.869.0>]
NPIds:[<0.871.0>,<0.872.0>]
APIds:[<0.870.0>]
ScapePids:[<0.867.0>]
Sensor:{{-1,7.57227593586992e-10},sensor} is terminating.
Agent:<0.866.0> terminating. Genotype has been backed up.
 Fitness:1.3179457789331406
 TotEvaluations:163
 TotCycles:656
 TimeAcc:23321
Cortex:{{origin,7.572275935869961e-10},cortex} is terminating.
```

It works! And from the highlighted NPIds, we can see the two spawned neuron PIds. The system terminated successfully, the topology we analyzed manually is correct given the mutation operator, and the phenotype works perfectly. Thus this mutation operator is functional, at least in this simple test, and we move on to the next one.

outsplice: This mutation operator selects a random neuron A in the NN, then selects the neuron's random output connection to some element B, disconnects A from B, creates a new neuron C in the layer between neuron A and element B (creating the new layer if it does not already exist, or using an existing one if A and B are one or more layers apart), and then reconnects A to B through C:

```
2> genotype:create_test().
...
{cortex,{{origin,7.57225527862836e-10},cortex},
      test,
      [{{0,7.572255278628331e-10},neuron}],
      [{{-1,7.572255278628343e-10},sensor}],
      [{{1,7.572255278628337e-10},actuator}]}
{sensor,{{-1,7.572255278628343e-10},sensor},
      xor_GetInput,
      {{origin,7.57225527862836e-10},cortex},
      {private,xor_sim},
      2,
      [{{0,7.572255278628331e-10},neuron}],
      undefined}
{neuron,{{0,7.572255278628331e-10},neuron},
      0,
      {{origin,7.57225527862836e-10},cortex},
      tanh,
      [{{{-1,7.572255278628343e-10},sensor},
        [0.4094174115111171,0.40477840576669655]}],
      [{{1,7.572255278628337e-10},actuator}],
      []}
{actuator,{{1,7.572255278628337e-10},actuator},
        xor_SendOutput,
        {{origin,7.57225527862836e-10},cortex},
        {private,xor_sim},
        1,
        [{{0,7.572255278628331e-10},neuron}],
        undefined}
{atomic,{atomic,[ok]}}
3> genome_mutator:test(test,outsplice).
{atomic,{atomic,ok}}
4> genotype:print(test).
...
{cortex,{{origin,7.57225527862836e-10},cortex},
      test,
      [{{0.5,7.572255205521553e-10},neuron},
       {{0,7.572255278628331e-10},neuron}],
      [{{-1,7.572255278628343e-10},sensor}],
      [{{1,7.572255278628337e-10},actuator}]}
{sensor,{{-1,7.572255278628343e-10},sensor},
      xor_GetInput,
      {{origin,7.57225527862836e-10},cortex},
      {private,xor_sim},
      2,
      [{{0,7.572255278628331e-10},neuron}],
      undefined}
{neuron,{{0.5,7.572255205521553e-10},neuron},
      0,
      {{origin,7.57225527862836e-10},cortex},
      absolute,
      [{{{0,7.572255278628331e-10},neuron},[0.08385901270641671]}],
      [{{1,7.572255278628337e-10},actuator}],
      []}
{neuron,{{0,7.572255278628331e-10},neuron},
      0,
      {{origin,7.57225527862836e-10},cortex},
      tanh,
      [{{{-1,7.572255278628343e-10},sensor},
        [0.4094174115111171,0.40477840576669655]}],
      [{{0.5,7.572255205521553e-10},neuron}],
      []}
{actuator,{{1,7.572255278628337e-10},actuator},
        xor_SendOutput,
        {{origin,7.57225527862836e-10},cortex},
        {private,xor_sim},
        1,
        [{{0.5,7.572255205521553e-10},neuron}],
        0}
{atomic,[ok]}
```

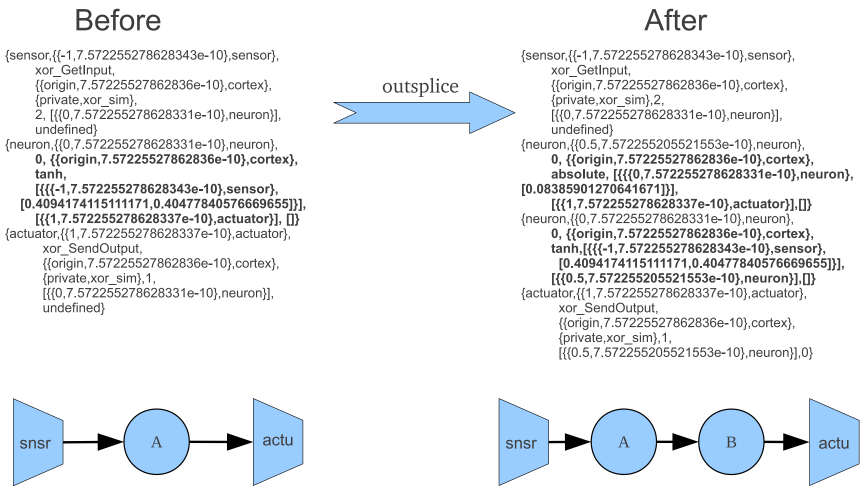
It works! The genotype:create_test() function created the genotype of a simple test NN system, with a single neuron:

```
{neuron,{{0,7.572255278628331e-10},neuron},
       0,
       {{origin,7.57225527862836e-10},cortex},
       tanh,
       [{{{-1,7.572255278628343e-10},sensor},
         [0.4094174115111171,0.40477840576669655]}],
       [{{1,7.572255278628337e-10},actuator}],
       []}
```

Which is connected from the sensor: **{{-1,7.572255278628343e-10},sensor}** and is connected to the actuator: **{{1,7.572255278628337e-10},actuator}**. From the neuron's Id, we can see that it is in layer 0. After we executed the outsplice mutation operator, our NN system acquired a new neuron, thus the NN now had two neurons:

```
{neuron,{{0.5,7.572255205521553e-10},neuron},
       0,
       {{origin,7.57225527862836e-10},cortex},
       absolute,
       [{{{0,7.572255278628331e-10},neuron},[0.08385901270641671]}],
       [{{1,7.572255278628337e-10},actuator}],
       []}
{neuron,{{0,7.572255278628331e-10},neuron},
       0,
       {{origin,7.57225527862836e-10},cortex},
       tanh,
       [{{{-1,7.572255278628343e-10},sensor},
         [0.4094174115111171,0.40477840576669655]}],
       [{{0.5,7.572255205521553e-10},neuron}],
       []}
```

Note that where as in the initial genotype the NN was composed of a single neuron: **{{0,7.572255278628331e-10}, neuron}**, which was connected from the sensor: **{{-1,7.572255278628343e-10}, sensor}**, and connected to the actuator: **{{1,7.572255278628337e-10}, actuator}**, after the mutation operator was applied, the NN acquired a new neuron, which was inserted into a new layer 0.5 (we determine that fact from its Id, which contains the layer index specification). Also note that the original neuron is no longer connected to the actuator, but instead is connected to the new neuron: **{{0.5,7.572255205521553e-10},neuron}**, which is now the one connected to the actuator. The diagram of the before and after topology of this NN system is shown in Fig-9.4.



```
{sensor,{{-1,7.572255278628343e-10},sensor},
      xor_GetInput,
      {{origin,7.57225527862836e-10},cortex},
      {private,xor_sim},
      2, [{{0,7.572255278628331e-10},neuron}],
      undefined}
{neuron,{{0,7.572255278628331e-10},neuron},
      0, {{origin,7.57225527862836e-10},cortex},
      tanh,
      [{{{-1,7.572255278628343e-10},sensor},
   [0.4094174115111171,0.40477840576669655]}],
      [{{1,7.572255278628337e-10},actuator}], []}
{actuator,{{1,7.572255278628337e-10},actuator},
        xor_SendOutput,
        {{origin,7.57225527862836e-10},cortex},
        {private,xor_sim},1,
        [{{0,7.572255278628331e-10},neuron}],
        undefined}
```

```
{sensor,{{-1,7.572255278628343e-10},sensor},
      xor_GetInput,
      {{origin,7.57225527862836e-10},cortex},
      {private,xor_sim},2,
      [{{0,7.572255278628331e-10},neuron}],
      undefined}
{neuron,{{0.5,7.572255205521553e-10},neuron},
      0, {{origin,7.57225527862836e-10},cortex},
      absolute, [{{{0,7.572255278628331e-10},neuron},
[0.0838590127064167l]}],
      [{{1,7.572255278628337e-10},actuator}],[]}
{neuron,{{0,7.572255278628331e-10},neuron},
      0, {{origin,7.57225527862836e-10},cortex},
      tanh,[{{{-1,7.572255278628343e-10},sensor},
        [0.4094174115111171,0.40477840576669655]}],
      [{{0.5,7.572255205521553e-10},neuron}],[]}
{actuator,{{1,7.572255278628337e-10},actuator},
        xor_SendOutput,
        {{origin,7.57225527862836e-10},cortex},
        {private,xor_sim},1,
        [{{0.5,7.572255205521553e-10},neuron}],0}
```

Fig. 9.4 The NN System topology before and after the outsplice mutation operator is applied to it.

Let's test this NN system by mapping its genotype to its phenotype, and applying it to the problem that its morphology defines (mimicking the XOR operator):

```
5> exoself:start(test,void).
<0.919.0>
Finished updating genotype
Terminating the phenotype:
Cx_PId:<0.921.0>
SPIds:[<0.922.0>]
NPIds:[<0.924.0>,<0.925.0>]
```

```
APIds:[<0.923.0>]
ScapePids:[<0.920.0>]
Agent:<0.919.0> terminating. Genotype has been backed up.
 Fitness:0.5311848171954074
 TotEvaluations:58
 TotCycles:236
 TimeAcc:7384
Cortex:{{origin,7.57225527862836e-10},cortex} is terminating.
Sensor:{{-1,7.572255278628343e-10},sensor} is terminating.
```

It works! And we can also see that there are two NPIds, since there are now two neurons. We have visually inspected the NN system genotype before and after the mutation operator was applied, and found the new genotype to be correct. We have also tested the phenotype, to ensure that it is functional, and confirmed that it is. We next test the two last remaining mutation operators: add_sensor and add_actuator.

add_sensor & add_actuator: The add_sensor mutation operator adds a new random sensor, still unused by the NN system. The sensor is chosen from the sensor list available to the morphology of the NN based agent. A random neuron in the NN is then chosen, and the sensor is connected to that neuron. The add_actuator mutation operator adds a new random actuator, still unused by the NN system. A random neuron in the NN is then chosen, and a link is established between this neuron and the new actuator.

```
2> genome_mutator:test(test,add_sensor).
{aborted,"********ERROR:add_sensor(Agent_Id):: NN system is already using all available
sensors"}
3> genome_mutator:test(test,add_actuator).
{aborted,"********ERROR:add_actuator(Agent_Id):: NN system is already using all available
actuators"}
```

This is as expected. The test NN system uses the xor_mimic morphology, and if we look in the morphology module, we see that it only has one sensor and one actuator. Thus, when we run the mutation operators for this particular test, our neuroevolutionary system does not add a new sensor, or a new actuator, because there are no new ones available. When we begin expanding the neuroevolutionary platform we're designing here, we will see the affects of a system that can incorporate new sensors and actuators into itself as it evolves. We can similarly test the mutation operators: add_sensorlink & add_actuatorlink, but just as the above two mutation operators, they have no new elements to connect to and from, respectively, when it comes to the seed NN.

We have now successfully tested most of the complexifying mutation operators on the simple, seed NN based agent. But this does not necessarily mean that there are no bugs in our system. Perhaps there are scenarios when it does fail, we just haven't come across them yet because we've only tested the operators on the most simple type of topology, the single neuron NN system topology.

Before we proceed, let's create a small program that applies X random mutation operators to the test NN system, and then converts the mutated genotype to its phenotype, to ensure that it still functions. The goal here is to ensure that the resulting NN is simply connected, and does not crash, or stall during operation. Furthermore, we can run this mutation operator test itself, a few thousand times. If at any point it gets stuck, or there is an unexpected error, we can then try to figure out what happened.

The following listing shows this simple, topological mutation testing function that we add to the genome_mutator module:

Listing-9.1 The long_test/1 function, which creates a seed agent, and applies TotMutateApplications number of mutation operators to it, and tests the resulting phenotype afterwards.

```
long_test(TotMutateApplications) when (TotMutateApplications > 0) ->
    genotype:create_test(),
    short_test(TotMutateApplications).

    short_test(0)->
            exoself:start(test,void);
    short_test(Index)->
            test(),
            short_test(Index-1).
%This is a simple function that executes the test() function the number of times with which the long_test/1 function was initially called. The test/0 function executes mutate(test), which applies a random number of mutation operators to the genotype, where that number ranges from 1 to sqrt(Tot_neurons). After all the mutation operators have been applied successfully, the function executes exoself:start(test,void), mapping the genotype to phenotype, to test whether the resulting NN system is functional.
```

The long_test/1 function will perform the following steps:

1. Create a test genotype.
2. Execute the mutate(test) function TotMutateApplications number of times.
3. Convert the genotype to phenotype to ensure that the resulting NN system is functional.

Lets run the *long_test* function with *TotMutateApplications = 300*. For the sake of being brief, I will only present the first and last few lines of the printout to console in the following Listing-9.2.

Listing-9.2 Running the long_test function, which applies a random number of mutation operators to the original seed agent, 300 times.

```
2>genome_mutator:long_test(300).
{agent,test,0,undefined,test,
    {{origin,7.571534416338085e-10},cortex},
    {[{0,1}],
     [],
     [{sensor,undefined,xor_GetInput,undefined,
          {private,xor_sim},
          2,
          [{{0,7.571534416338051e-10},neuron}],
          undefined}],
     [{actuator,undefined,xor_SendOutput,undefined,
           {private,xor_sim},
           1,
           [{{0,7.571534416338051e-10},neuron}],
           undefined}]},
    {constraint,xor_mimic,[tanh,cos,gaussian,absolute]},
    [],undefined,0,
    [{0,[{{0,7.571534416338051e-10},neuron}]}]}
…
Tot neurons:1 Performing Tot mutations:1 on:test
Mutation Operator:add_outlink
******** Mutation Succesful.
Tot neurons:1 Performing Tot mutations:1 on:test
Mutation Operator:add_actuator
******** Error:{aborted,"********ERROR:add_actuator(Agent_Id):: NN system is already
using all available actuators"}
Retrying with new Mutation...
Mutation Operator:outsplice
******** Mutation Succesful.
Tot neurons:2 Performing Tot mutations:1 on:test
Mutation Operator:mutate_af
******** Mutation Succesful.
...
Tot neurons:95 Performing Tot mutations:5 on:test
Mutation Operator:outsplice
Mutation Operator:add_bias
Mutation Operator:mutate_weights
Mutation Operator:add_outlink
Mutation Operator:mutate_af
******** Mutation Succesful.
<0.2460.0>
Finished updating genotype
```

```
Terminating the phenotype:
Cx_PId:<0.2463.0>
SPIds:[<0.2464.0>]
NPIds:[<0.2467.0>,<0.2468.0>,<0.2469.0>,<0.2470.0>,<0.2471.0>,<0.2472.0>,<0.2473.0>,
<0.2474.0>,<0.2475.0>,<0.2476.0>,<0.2477.0>,<0.2478.0>,<0.2479.0>,<0.2480.0>,
<0.2481.0>,<0.2482.0>,<0.2483.0>,<0.2484.0>,<0.2485.0>,<0.2486.0>,<0.2487.0>,
<0.2488.0>,<0.2489.0>,<0.2490.0>,<0.2491.0>,<0.2492.0>,<0.2493.0>,<0.2494.0>,
<0.2495.0>,<0.2496.0>,<0.2497.0>,<0.2498.0>,<0.2499.0>,<0.2500.0>,<0.2501.0>,
<0.2502.0>,<0.2503.0>,<0.2504.0>,<0.2505.0>,<0.2506.0>,<0.2507.0>,<0.2508.0>,
<0.2509.0>,<0.2510.0>,<0.2511.0>,<0.2512.0>,<0.2513.0>,<0.2514.0>,<0.2515.0>,
<0.2516.0>,<0.2517.0>,<0.2518.0>,<0.2519.0>,<0.2520.0>,<0.2521.0>,<0.2522.0>,
<0.2523.0>,<0.2524.0>,<0.2525.0>,<0.2526.0>,<0.2527.0>,<0.2528.0>,<0.2529.0>,
<0.2530.0>,<0.2531.0>,<0.2532.0>,<0.2533.0>,<0.2534.0>,<0.2535.0>,<0.2536.0>,
<0.2537.0>,<0.2538.0>,<0.2539.0>,<0.2540.0>,<0.2541.0>,<0.2542.0>,<0.2543.0>,
<0.2544.0>,<0.2545.0>,<0.2546.0>,<0.2547.0>,<0.2548.0>,<0.2549.0>,<0.2550.0>,
<0.2551.0>,<0.2553.0>,<0.2554.0>,<0.2555.0>,<0.2556.0>,<0.2557.0>,<0.2558.0>,
<0.2559.0>,<0.2560.0>,<0.2561.0>,<0.2562.0>,<0.2563.0>]
APIds:[<0.2465.0>,<0.2466.0>]
ScapePids:[<0.2461.0>,<0.2462.0>]
Sensor:{{-1,7.57153413903982e-10},sensor} is terminating.
Agent:<0.2460.0> terminating. Genotype has been backed up.
 Fitness:0.5162814284277237
 TotEvaluations:65
 TotCycles:132
 TimeAcc:21664
Cortex:{{origin,7.571534139039844e-10},cortex} is terminating.
```

From the above console printout, you can see that the first mutation operator applied was the add_outlink, which was successful. The second was add_actuator, which was not. At this stage, every time the mutate(test) gets executed, the function only applies a single mutation operator to the genotype, we know this from the line: **Tot neurons:1 Performing Tot mutations:1 on:test**. We then skip to the end, the last execution of the mutate(test). From the line: **Tot neurons:95 Performing Tot mutations:5 on:test**, we can see that at this point the NN system has 95 neurons, and the randomly chosen number of mutation operators to be applied is 5. This means that 5 mutation operators are applied in series to the NN system to produce the mutant agent, and only after the 5 mutation operators are applied, is the agent's fitness evaluated.

Once all the mutation operators have been applied, the exoself converts the genotype of the test NN system to its phenotype, applying it to the problem that its morphology designated it for. From the console printout, we see that the NN system successfully terminated, and so we can be assured that the NN topology does not have any discontinuities, and that it does produce a functional, albeit not very fit, phenotype. Also, none of the mutation operators produced any type of errors that originate from actual crashes.

Having now tested the main mutation operators and the mapping from genotype to phenotype, we can move on and see if the population_monitor is functional, by running the small XOR based benchmark, as we did in Chapter-7.

9.2 Testing the Neuroevolutionary System on the Simple XOR Benchmark

Having now tested some of the important independent functions and elements of our topology and weight evolving artificial neural network (TWEANN) system, we can move on to testing the system as a whole. Our morphology module contains various morphologies at our disposal, where a morphology is a list of sensors and actuators that a NN system can incorporate through evolution if it is of that particular morphology. Furthermore, the sensors and actuators define what the NN system can interface with, what the NN system does, and thus, what the problems the NN system is applied to. For example, if the sensor available to our NN system is one that reads values from a database, and the actuator is one that simply outputs the NN's output vector signal, and furthermore the database from which the sensor reads its data is a XOR truth table, then we could train this NN system to mimic a XOR logic operator. We could compare the NN based agent's output to what that output should be if the agent was a XOR logic operator, rewarding it if it's output is similar to the expected XOR operator output, and punishing it if not.

If the sensors were to have been programs that interfaced with a simulated world through sensors embedded in some simulated organism inhabiting a simulated world, and if the actuators were to have been programs controlling the simulated organism (avatar), then our NN system would be the evolving brain of an organism in an Artificial Life experiment. Thus, the sensors and actuators define what the NN system does, and its morphology is a set of sensors and actuators, as a package, available to the NN system during its evolution. Thus it is the morphology that defines the problem to what the NN system is applied. We choose a morphology to which the NN system belongs, and it evolves and learns how to use the sensors and actuators belonging to that morphology.

Thus far we have only created one morphology, the *xor_mimic*. The xor_mimic morphology contains a single sensor with the name *xor_GetInput*, and a single actuator with the name *xor_SendOutput*. Thus if we evolve agents of this particular morphology, they will only be able to evolve into XOR logical operator mimics. Agents cannot switch morphologies mid-evolution, but new sensors and actuators can be added to the morphology by updating the morphology module, and afterwards these new interfaces can then be incorporated into the NN system over time.

We created the population_monitor process which creates a seed population of NN systems belonging to some specified morphologies, and then evolves those NN based agents. Since the morphologies define the scapes the NN system interfaces with, and the scape computes the fitness score of the agent interfacing with it, the population_monitor process has the ability to evolve the population by having access to each agent's fitness in the population, applying a selection function to the population, and then mutating the selected agents, creating new and mutated offspring from them. We now test this process by getting the population_monitor process to spawn a seed population of agents with the xor_mimic morphology, and see how quickly our current version of the neuroevolutionary system can evolve a solution to this problem, how quickly it can evolve a XOR logic operator using neurons as the basic elements of the evolving network.

We will run the population_monitor:test() function with the following parameters:

```
%%%%%%%%%%%%%% Population Monitor Options & Parameters %%%%%%%%%%%%
-define(SELECTION_ALGORITHM,competition).
-define(EFF,0.2).
-define(INIT_CONSTRAINTS,[#constraint{morphology=Morphology, neu-
ral_afs=Neural_AFs}|| Morphology<-[xor_mimic],Neural_AFs<-[[tanh]]]).
-define(SURVIVAL_PERCENTAGE,0.5).
-define(SPECIE_SIZE_LIMIT,10).
-define(INIT_SPECIE_SIZE,10).
-define(INIT_POPULATION_ID,test).
-define(OP_MODE,gt).
-define(INIT_POLIS,mathema).
-define(GENERATION_LIMIT,100).
-define(EVALUATIONS_LIMIT,100000).
-define(DIVERSITY_COUNT_STEP,500).
-define(GEN_UID,genotype:generate_UniqueId()).
-define(CHAMPION_COUNT_STEP,500).
-define(FITNESS_GOAL,inf).
```

The population will thus be composed of NN systems using the xor_mimic morphology (and thus be applied to that particular problem), and whose neurons will use only the tanh activation function. The population will maintain a size close to 10. Finally, neuroevolution will continue for at most 100 generations, or at most 100000 evaluations. The fitness goal is set to inf, which means that it is not a stopping condition and the evolution will continue until one of the other terminating conditions is reached. The fitness score for each agent is calculated by the scape it is interfacing with. Having set up the parameters for our neuroeovlutionary system, we compile the population_monitor module, and execute the population_monitor:test() function, as shown next:

```
2> population_monitor:test().
Specie_Id:7.570104741922324e-10 Morphology:xor_mimic
******** Population monitor started with parameters:{gt,test,competition}
…
Selection Algorirthm:competition
Valid_AgentSummaries:[{91822.42396111514,3,{7.570065786458927e-10,agent}},
                {82128.75594984594,3,{7.570065785419657e-10,agent}},
                {66717.38827549343,3,{7.570065785184491e-10,agent}},
                {66865.26402662563,4,{7.570065786995862e-10,agent}},
                {66859.35543290272,4,{7.570065785258691e-10,agent}},
                {60974.864233884604,4,{7.570065785388116e-10,agent}}]
Invalid_AgentSummaries:[{56725.927279906005,4,{7.570065787547878e-10,agent}},
                 {46423.91939090131,4,{7.570065786090063e-10,agent}},
                 {34681.35604691528,3,{7.570065790439459e-10,agent}},
                 {67.37546054504678,4,{7.570065785110257e-10,agent}},
                 {13.178830126581289,5,{7.570065785335377e-10,agent}}]
NeuralEnergyCost:13982.434363128335
NewPopAcc:9.218546902348272
Population size normalizer:0.9218546902348272
Agent_Id:{7.570065785388116e-10,agent} Normalized_MutantAlotment:1
Agent_Id:{7.570065785258691e-10,agent} Normalized_MutantAlotment:1
Agent_Id:{7.570065786995862e-10,agent} Normalized_MutantAlotment:1
Agent_Id:{7.570065785184491e-10,agent} Normalized_MutantAlotment:2
...
******** Population_Monitor:test shut down with Reason:normal OpTag:continue, while in
OpMode:gt
******** Tot Agents:9 Population Generation:100 Eval_Acc:63960 Cycle_Acc:217798
Time_Acc:12912953
```

It works! Highlighted in green (2nd and 3rd line in the black & white printed version) are the first two lines printed to screen after population_monitor:test() is executed. It states that the population_monitor is started with selection algorithm *competition*, a population with the id *test*, and op_mode (operational mode) being gt, whose operational importance we will set in a later chapter.

Based on how we designed our population_monitor system, every generation it prints out the fitness score of the population. Highlighted in red and italicized is the 100th generation, and each agent with its fitness score. The most fit agent with its fitness score in the last generation is: {91822.42396111514, 3, {7.570065786458927e-10, agent}}. Based on how the xor_sim scape calculates fitness, this fitness score amounts to the agent having a mean squared sum error of 1/91822, and it took a total of 63960 evaluations for our neuroevolutionary system to reach it.

This is quite a bit of computational time for such a simple problem, but it is not usually the case to take the circuit to this level of accuracy. Let us change the fitness goal to 1000, make MAX_ATTEMPTS = 10 in the exoself module, and then try again.

In my experiment, I had the following results:

```
Valid_AgentSummaries:[{1000.4594763865106,2,{7.570051345044739e-10,agent}},
                {272.7339484226029,2,{7.570051345273578e-10,agent}},
                {249.64913390960575,2,{7.57005134500996e-10,agent}},
                {227.8298020262745 6,4,{7.570051345098297e-10,agent}},
                {193.32888692741093,2,{7.570051345440797e-10,agent}}]
Invalid_AgentSummaries:[{56.2580273824466,2,{7.570051346068126e-10,agent}},
                 {18.43287953405122,2,{7.570051345575052e-10,agent}},
                 {6.153281918877250 5,2,{7.570051345123884e-10,agent}},
                 {0.49999782678670823,3,{7.570051345394602e-10,agent}}]
…
…
…
******** Population_Monitor:test shut down with Reason:normal OpTag:continue, while in
OpMode:gt
******** Tot Agents:9 Population Generation:78 Eval_Acc:10701 Cycle_Acc:41178
Time_Acc:2259258
```

This time it took only 10701 evaluations. But there is something very interesting that happened here. Take a look at the most fit agent in the population, with the id: **{1000.4594763865106,2,{7.570051345044739e-10,agent}}**. It only has 2 neurons! That's not possible, since this particular circuit requires at least 3 neurons, if those neurons are using tanh activation function. We have the agent's Id, let's check out its topology, as shown in the following listing:

Listing-9.3 The console printout of the topology of the fittest agent in the population.

```
3> genotype:print({7.570051345044739e-10,agent}).
{agent,{7.570051345044739e-10,agent},
     15,undefined,7.570051363681182e-10,
     {{origin,7.570051345042693e-10},cortex},
     {[{0,1},{0.5,1}],
      [{add_bias,{0.5,neuron}},
       {mutate_af,{0,neuron}},
       {mutate_weights,{0.5,neuron}},
       {add_actuator,{0,neuron},{1,actuator}},
       {outsplice,{0,neuron},{0.5,neuron},{1,actuator}},
       {mutate_weights,{0,neuron}},
       {mutate_af,{0,neuron}},
       {mutate_af,{0,neuron}},
       {mutate_weights,{0,neuron}},
       {mutate_af,{0,neuron}},
       {mutate_weights,{0,neuron}},
       {mutate_af,{0,neuron}},
       {mutate_weights,{0,neuron}},
       {add_bias,{0,neuron}},
       {add_inlink,{0,neuron},{0,neuron}}],
      [{sensor,undefined,xor_GetInput,undefined,
            {private,xor_sim},
            2,
            [{{0,7.570051345042682e-10},neuron}],
            undefined}],
      [{actuator,undefined,xor_SendOutput,undefined,
             {private,xor_sim},
             1,
             [{{0.5,7.570051345042677e-10},neuron}],
             11},
       {actuator,undefined,xor_SendOutput,undefined,
             {private,xor_sim},
             1,
             [{{0,7.570051345042682e-10},neuron}],
             undefined}]},
     {constraint,xor_mimic,[tanh]},
     [{add_bias,{{0.5,7.570051345042677e-10},neuron}},
      {mutate_af,{{0,7.570051345439552e-10},neuron}},
      {mutate_weights,{{0.5,7.570051346065783e-10},neuron}},
      {add_actuator,{{0,7.57005134638638e-10},neuron},
             {{1,7.57005134636634e-10},actuator}},
      {outsplice,{{0,7.57005134670089e-10},neuron},
             {{0.5,7.570051346689715e-10},neuron},
             {{1,7.570051346700879e-10},actuator}},
      {mutate_weights,{{0,7.570051347808065e-10},neuron}},
      {mutate_af,{{0,7.570051347949999e-10},neuron}},
      {mutate_af,{{0,7.570051348731883e-10},neuron}},
      {mutate_weights,{{0,7.57005134905699e-10},neuron}},
      {mutate_af,{{0,7.570051352005185e-10},neuron}},
      {mutate_weights,{{0,7.57005135384367e-10},neuron}},
      {mutate_af,{{0,7.570051357421974e-10},neuron}},
      {mutate_weights,{{0,7.570051357953169e-10},neuron}},
      {add_bias,{{0,7.570051361212367e-10},neuron}},
      {add_inlink,{{0,7.570051363350866e-10},neuron},
             {{0,7.570051363350866e-10},neuron}}],
     1000.4594763865106,0,
     [{0,[{{0,7.570051363631578e-10},neuron}]},
        {0.5,[{{0.5,7.570051346689715e-10},neuron}]}]}
{cortex,{{origin,7.570051345042693e-10},cortex},
        {7.570051345044739e-10,agent},
        [{{0.5,7.570051345042677e-10},neuron},
         {{0,7.570051345042682e-10},neuron}],
        [{{-1,7.570051345042671e-10},sensor}],
        [{{1,7.570051345042659e-10},actuator},
         {{1,7.570051345042664e-10},actuator}]}
{sensor,{{-1,7.570051345042671e-10},sensor},
        xor_GetInput,
        {{origin,7.570051345042693e-10},cortex},
        {private,xor_sim},
        2,
        [{{0,7.570051345042682e-10},neuron}],
        undefined}
{neuron,{{0.5,7.570051345042677e-10},neuron},
        15,
        {{origin,7.570051345042693e-10},cortex},
        tanh,
        [{{{0,7.570051345042682e-10},neuron},[-4.9581978771372395]},
         {bias,[-2.444318048832683]}],
        [{{1,7.570051345042659e-10},actuator}],
        []}
{neuron,{{0,7.570051345042682e-10},neuron},
        14,
        {{origin,7.570051345042693e-10},cortex},
        tanh,
        [{{{0,7.570051345042682e-10},neuron},[6.283185307179586]},
         {{{-1,7.570051345042671e-10},sensor},
          [-4.3985975891263305,-2.3223009779757877]},
         {bias,[1.3462974501315348]}],
        [{{1,7.570051345042664e-10},actuator},
         {{0.5,7.570051345042677e-10},neuron},
         {{0,7.570051345042682e-10},neuron}],
        [{{0,7.570051345042682e-10},neuron}]}
{actuator,{{1,7.570051345042659e-10},actuator},
          xor_SendOutput,
          {{origin,7.570051345042693e-10},cortex},
          {private,xor_sim},
          1,
          [{{0.5,7.570051345042677e-10},neuron}],
          11}
{actuator,{{1,7.570051345042664e-10},actuator},
          xor_SendOutput,
          {{origin,7.570051345042693e-10},cortex},
          {private,xor_sim},
          1,
          [{{0,7.570051345042682e-10},neuron}],
          undefined}
{atomic,[ok,ok]}
```

Though we've decided to look at the NN system's genotype to see how it was possible for our neuroevolutionary system to evolve a solution with only two neurons, instead, if you look through the genotype, you will see that we just uncovered a large number of errors in the way our system functions. Let's take a look at each part in turn, before returning to the actual evolved topology of the NN system.

Boldfaced in the console printout above are the following errors, discussed and corrected in the following sections:

1. mutate_af operator is applied to the agent multiple times, but we have opted to only use the tanh activation function, which means this mutation operator does nothing to the network, and is a waste of a mutation attempt, and thus should not be present.
2. When looking at the mutate_af, we also see that it is applied to neurons with different Ids, 5 of them, even though there are only 2 neurons in the system.
3. This NN system evolved a connection to two actuators, but this morphology supports only 1, what happened?
4. In the agent's fingerprint, the sensors and actuators contain N_Ids. This is an error, since the fingerprint must not contain any Id specific information, it must only contain the general information about the NN system, so that we can have an ability to roughly distinguish between different species of the NN systems (those with different topologies, morphologies, sensors and actuators, or those with significantly different sets of activation functions).

In the following sections, we deal with each of these errors one at a time.

9.2.1 The mutate_af Error

Looking at the agent's evo_hist list, shown in Listing-9.4, we can see that multiple mutate_afs are applied. The goal of a mutation operator is to modify the NN system, and if a mutation operator cannot be applied, due to for example the state in which the NN system is, or because it leads to a non-functional NN, then we should revert the mutation operator and try applying another one. Each NN system, when being mutated, undergoes a specific number of mutations, ranging from 1 to sqrt(Tot_Neurons), chosen randomly. Thus, every time we apply a mutation operator to the NN system, and it does nothing, that is one mutation attempt wasted. This can result in a clone which was not mutated at all, or not mutated properly.

Afterwards, this clone is sent back into the environment to be evaluated. For example assume a fit agent creates an offspring by first creating a clone of itself, and then applying to it the mutate_af operator, if mutate_af is being applied to an agent that only has tanh for its available activation functions list, the resulting offspring is exactly the same as its parent, since tanh was swapped for tanh. There is no reason to test out a clone, since we already know how such a NN system functions, because its parent has already been evaluated and tested for fitness. It is thus essential that whatever is causing this error, is fixed.

Listing-9.4 The agent's evo_hist list.

```
 [{add_bias,{{0.5,7.570051345042677e-10},neuron}},
       {mutate_af,{{0,7.570051345439552e-10},neuron}},
       {mutate_weights,{{0.5,7.570051346065783e-10},neuron}},
       {add_actuator,{{0,7.57005134638638e-10},neuron},
                 {{1,7.57005134636634e-10},actuator}},
       {outsplice,{{0,7.57005134670089e-10},neuron},
               {{0.5,7.570051346689715e-10},neuron},
               {{1,7.570051346700879e-10},actuator}},
       {mutate_weights,{{0,7.570051347808065e-10},neuron}},
       {mutate_af,{{0,7.570051347949999e-10},neuron}},
       {mutate_af,{{0,7.570051348731883e-10},neuron}},
       {mutate_weights,{{0,7.57005134905699e-10},neuron}},
       {mutate_af,{{0,7.570051352005185e-10},neuron}},
       {mutate_weights,{{0,7.57005135384367e-10},neuron}},
       {mutate_af,{{0,7.570051357421974e-10},neuron}},
       {mutate_weights,{{0,7.570051357953169e-10},neuron}},
       {add_bias,{{0,7.570051361212367e-10},neuron}},
       {add_inlink,{{0,7.570051363350866e-10},neuron},
                {{0,7.570051363350866e-10},neuron}}],
```

To solve this problem we need to check the *genome_mutator:mutate_af/1* function, as shown in listing-9.5.

Listing-9.5 The mutate_af/1 function.

```
mutate_af(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}),
   AF = N#neuron.af,
   Activation_Functions = (A#agent.constraint)#constraint.neural_afs -- [AF],
   NewAF = genotype:generate_NeuronAF(Activation_Functions),
   U_N = N#neuron{af=NewAF,generation=Generation},
   EvoHist = A#agent.evo_hist,
   U_EvoHist = [{mutate_af,N_Id}|EvoHist],
   U_A = A#agent{evo_hist=U_EvoHist},
   genotype:write(U_N),
   genotype:write(U_A).
```

Though: *Activation_Functions = (A#agent.constraint)#constraint.neural_afs – [AF]*, does result in an empty list (since #constraint.neural_afs list is: [tanh]), it does not matter because the *genotype:generate_NeuronAF(Activation_Functions)* function itself chooses the default *tanh* activation function when executed with an empty list parameter. This is the cause of this error. What we need to do is simply exit the mutation operator as soon as we find that there is only one activation function, that it is already being used by the neuron, and that there is nothing to mutate. We thus modify mutate_af/1 function to be as follows:

Listing-9.6 The mutate_af function after the fix is applied.

```
mutate_af(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}),
    AF = N#neuron.af,
    case (A#agent.constraint)#constraint.neural_afs -- [AF] of
            [] ->
                    exit("********ERROR:mutate_af:: There are no other activation func-
tions to use.");
            Activation_Functions ->
                    NewAF = lists:nth(random:uniform(length(Activation_Functions)),
Activation_Functions),
                    U_N = N#neuron{af=NewAF,generation=Generation},
                    EvoHist = A#agent.evo_hist,
                    U_EvoHist = [{mutate_af,N_Id}|EvoHist],
                    U_A = A#agent{evo_hist=U_EvoHist},
                    genotype:write(U_N),
                    genotype:write(U_A)
    end.
```

The fix is shown in boldface. In this fixed function, as soon as the mutation operator determines that there are no other activation functions that it can swap the currently used one for, it simply exits with an error. The genome mutator then tries out another mutation operator.

9.2.2 Same Neuron, But Different Ids in the evo_hist List

Looking at the Listing-9.3 again, we also see that even though the NN has only 2 neurons, as was shown in the original printout to console, there were 5 mutate_af operators applied and each one was applied to a different neuron_id. But how is that possible if there are only 2 neurons and thus only 2 different neuron ids?

This error occurs because when we clone the NN, all neurons get a new id, but we never update the evo_hist list, converting those old ids into new ones. This means that the Ids within the evo_hist are not of the elements belonging to the agent in its current state, but the element ids which belong to its ancestors. Though it does not matter what the particular ids are, it is essential that they are consistent, so that we can reconstruct the evolutionary path of the NN based system, which is not possible if we don't know which mutation operator was applied to which element in the NN system being analyzed. To be able to see when, and to what particular elements of the topology the mutation operators were applied, we need a consistent set of element ids in the evo_hist, so that the evolutionary path can be reconstructed based on the actual ids used by the NN based agent.

To fix this, we need to modify the cloning process so that it does not only update all the element ids in the NN system, but also the element ids in the evo_hist, ensuring that the system is consistent. The cloning process is performed in the genotype module, through the clone_Agent/2 function. Therefore, it is this function that we need to correct. The fix is simple, we need to create a new function called map_EvoHist/2, and call it from the clone_Agent/2 function with the old evo_hist list and an ETS table containing a map from old ids to new ones. The map_EvoHist/2 function can then map the old ids to new ids in the evo_hist list. The cloned agent will then use this updated evo_hist, with its updated new ids, instead of the old ids which belonged to its parent. The updated map_EvoHist/2 function is shown in Listing-9.7.

Listing-9.7 A new function, map_EvoHist/2, which updates the element ids of the evo_hist list, mapping the ids of the original agent to the ids of the elements used by its clone.

```
map_EvoHist(TableName,EvoHist)->
   map_EvoHist(TableName,EvoHist,[]).

map_EvoHist(TableName,[{MO,E1Id,E2Id,E3Id}|EvoHist],Acc)->
```

```
    Clone_E1Id = ets:lookup_element(TableName,E1Id,2),
    Clone_E2Id = ets:lookup_element(TableName,E2Id,2),
    Clone_E3Id = ets:lookup_element(TableName,E3Id,2),
    map_EvoHist(TableName,EvoHist,[{MO,Clone_E1Id,Clone_E2Id, Clone_E3Id}| Acc]);
map_EvoHist(TableName,[{MO,E1Id,E2Id}|EvoHist],Acc)->
    Clone_E1Id = ets:lookup_element(TableName,E1Id,2),
    Clone_E2Id = ets:lookup_element(TableName,E2Id,2),
    map_EvoHist(TableName,EvoHist,[{MO,Clone_E1Id,Clone_E2Id}|Acc]);
map_EvoHist(TableName,[{MO,E1Id}|EvoHist],Acc)->
    Clone_E1Id = ets:lookup_element(TableName,E1Id,2),
    map_EvoHist(TableName,EvoHist,[{MO,Clone_E1Id}|Acc]);
map_EvoHist(_TableName,[],Acc)->
    lists:reverse(Acc).
%map_EvoHist/2 is a wrapper for map_EvoHist/3, which in turn accepts the evo_hist list con-
taining the mutation operator tuples that have been applied to the NN system. The function is
used when a clone of a NN system is created. The function updates the original Ids of the ele-
ments the mutation operators have been applied to, to the ids used by the elements of the clone,
so that the updated evo_hist can reflect the clone's topology, as if all the mutation operators
have been applied to it instead, and that it is not a clone. Once all the tuples in the evo_hist have
been updated with the clone's element ids, the list is reversed to its proper order, and the updat-
ed list is returned to the caller.
```

Having fixed this bug, we move on to the next one.

9.2.3 Multiple Actuators of the Same Type

Looking again at the Listing-9.3, we see that one of the mutation operators was *add_actuator*. Since only successful mutation operators are allowed to be in the *evo_hist* list, it must be the case that only those mutation operators that actually mutated the genotype are present in the evo_hist list, which is what allows us to use it to trace back the evolutionary path of the evolved agent. But the presence of add_actuator in evo_hist must be an error, because the xor_mimic morphology only gives the agent access to a single actuator, there are no variations of that actuator, and the agent starts with that single actuator. It should not be possible to add a new actuator to the NN system since there are no new ones available, and this tag should not exist in the evo_hist list. This mutation operator was applied in error, let's find out why.

Looking at the add_actuator/1 in the genome_mutator module, we can see that *it does* check whether all the actuators are already used. But if we look at the agent's fingerprint section of the console printout in Listing-9.3:

```
[{actuator,undefined,xor_SendOutput,undefined,
            {private,xor_sim},
            1,
            [{{0.5,7.570051345042677e-10},neuron}],
            11},
    {actuator,undefined,xor_SendOutput,undefined,
            {private,xor_sim},
            1,
            [{{0,7.570051345042682e-10},neuron}],
            undefined}]
```

We notice the problem. The last element in the record defining the actuator in the genotype is the generation element. One actuator has the generation set to *11*, the other has it set to *undefined*. In the add_actuator function, we do not reset the *generation* value, as we do with the *id* and the *cx_id*. This must be it. When we subtract the list of the actuators used by the agent from the morphology's list of available actuators, the resulting list is not empty. The reason why an actuator still remains in the list, is because we did not set the generation parameter of the agent's actuator to undefined. Since the two actuators are not exactly the same (with all their agent specific features been set to defaults), the actuator used by the agent is not removed from the list of available actuators of the morphology's actuator list.

This also raises the issue of what should we do, in a consistent manner, with the generation parameter of the actuator? Lets update the source code and treat the generation of the actuator element as we treat it in the neuron elements: Initially set it to the value of the generation when it was created, and update its value every time it has been *affected* by a mutation. We make the same modification to the sensor elements.

In the add_actuator/1 function we change the line:

```
...
case morphology:get_Actuators(Morphology)--[(genotype:read({actuator, A_Id}))#actuator{
cx_id=undefined,  id=undefined, fanin_ids=[]} || A_Id<-A_Ids] of
...
```

To:

```
…
case morphology:get_Actuators(Morphology)--[(genotype:read({actuator, A_Id}))#actuator{
cx_id=undefined, id=undefined, fanin_ids=[],generation=undefined} || A_Id<-A_Ids] of
…
```

We do the same thing to the add_sensor/1 function. And then to ensure that the actuator's generation is updated every time a mutation operator affects it, we update the function linkFromNeuronToActuator/3 from using the line:

```
    genotype:write(ToA#actuator{ fanin_ids=U_Fanin_Ids})
```

To one using:

```
    genotype:write(ToA#actuator{ fanin_ids = U_Fanin_Ids, generation=Generation})
```

To make sure that the sensor's generation is also updated, we modify the function *link_FromSensorTo/2* from:

```
link_FromSensor(FromS,ToId)->
    FromFanout_Ids = FromS#sensor.fanout_ids,
    case lists:member(ToId, FromFanout_Ids) of
            true ->
                    exit("******** ERROR:link_FromSensor[cannot add ToId to Sensor]: ~p al-
ready a member of ~p~n",[ToId,FromS#sensor.id]);
            false ->
                    FromS#sensor{
                            fanout_ids = [ToId|FromFanout_Ids]
                    }
    end.
```

To the function *link_FromSensorTo/3*:

```
link_FromSensor(FromS,ToId,Generation)->
    FromFanout_Ids = FromS#sensor.fanout_ids,
    case lists:member(ToId, FromFanout_Ids) of
            true ->
                    exit("******** ERROR:link_FromSensor[can not add ToId to Sensor]: ~p al-
ready a member of ~p~n",[ToId,FromS#sensor.id]);
            false ->
                    FromS#sensor{
                            fanout_ids = [ToId|FromFanout_Ids],
                            generation=Generation
                    }
    end.
```

Finally, we also update the genotype module's function *construct_Cortex/3*, from using:

```
    Sensors = [S#sensor{id={{-1,generate_UniqueId()},sensor},cx_id=Cx_Id}|| S<-
morphology:get_InitSensors(Morphology)],
    Actuators = [A#actuator{id={{1,generate_UniqueId()},actuator},cx_id=Cx_Idn}||A<-
morphology:get_InitActuators(Morphology)],
```

To one using:

```
    Sensors = [S#sensor{id={{-1,generate_UniqueId()},sensor},cx_id=Cx_Id,
generation=Generation} || S<- morphology:get_InitSensors(Morphology)],
    Actuators = [A#actuator{id={{1,generate_UniqueId()},actuator},cx_id=Cx_Id,
generation=Generation} || A<-morphology:get_InitActuators(Morphology)],
```

Which ensures that we can keep track of the generation from the very start.

9.2.4 Making Fingerprint Store Generalized Sensors & Actuators

The fingerprint of the agent is used to vaguely represent the species that the agent belongs to. For example, if we have two NN systems which are exactly the same, except for the ids of their elements and the synaptic weights their neurons use, then these two agents belong to the same species. We cannot compare them directly to each other, because they will have those differences (the ids and the synaptic weights), but we can create a more generalized fingerprint for each agent which will be exactly the same for both. Some of the general features which we might use to classify a species is the NN topology and the sensors and actuators the NN system uses.

The 4th error we noticed was that we forgot to get rid of the N_Ids in the *generalized* sensor and actuator tuples within the fingerprint. We got rid of all the Id specific parts (the element's own id, and the cx_id) of those tuples before entering them into the fingerprint tuple, but forgot to do the same for the fanin_ids and the fanout_ids in the actuator and sensor tuples respectively. The fix is very simple, in the genotype module, we modify two lines in the update_fingerprint/1 function from:

```
    GeneralizedSensors= [(read({sensor,S_Id}))#sensor{id=undefined,cx_id=undefined}
|| S_Id<-Cx#cortex.sensor_ids],
    GeneralizedActuators= [(read({actuator,A_Id}))#actuator{id=undefined, cx_id=undefined}
|| A_Id<-Cx#cortex.actuator_ids],
```

To:

```
    GeneralizedSensors= [(read({sensor,S_Id}))#sensor{id=undefined,cx_id=undefined,
fanout_ids =[]} || S_Id<-Cx#cortex.sensor_ids],
    GeneralizedActuators= [(read({actuator,A_Id}))#actuator{id=undefined,cx_id =undefined,
fanin_ids=[]} || A_Id<-Cx#cortex.actuator_ids],
```

This change fixes the 4th and final error we've noticed. With this done, we now take our attention towards the remaining noticed anomaly, the 2 neuron NN solution. How is it possible?

9.2.5 The Quizzical Topology of the Fittest NN System

The first thing we noticed, and the reason for a closer analysis of the evolved agent, was the NN's topology, the fact that it had 2 neurons instead of 3+ neurons. After our analysis though, and finding out that it also had 2 actuators, while interfacing with only a single private scape, which means that both actuators were sending signals to it... there might be all kinds of different reasons for the 2 neuron solution. Nevertheless, let us still build it to see what exactly has evolved. Fig-9.5 shows the diagram of the final evolved NN system, based on the genotype in Listing-9.3.

```
{cortex,{{origin,7.570051345042693e-10},cortex},
        {7.570051345044739e-10,agent},
        [{{0.5,7.570051345042677e-10},neuron},
         {{0,7.570051345042682e-10},neuron}],
        [{{-1,7.570051345042671e-10},sensor}],
        [{{1,7.570051345042659e-10},actuator},
         {{1,7.570051345042664e-10},actuator}]}
{sensor,{{-1,7.570051345042671e-10},sensor},
        xor_GetInput,
        {{origin,7.570051345042693e-10},cortex},
        {private,xor_sim}, 2,
        [{{0,7.570051345042682e-10},neuron}],
        undefined}
{neuron,{{0.5,7.570051345042677e-10},neuron}, 15,
        {{origin,7.570051345042693e-10},cortex},
        tanh,
        [{{{0,7.570051345042682e-10},neuron},[-4.9581978771372395]},
         {bias,[-2.444318048832683]}],
        [{{1,7.570051345042659e-10},actuator}],
        []}
{neuron,{{0,7.570051345042682e-10},neuron}, 14,
        {{origin,7.570051345042693e-10},cortex},
        tanh,
        [{{{0,7.570051345042682e-10},neuron},[6.283185307179586]},
         {{{-1,7.570051345042671e-10},sensor},
         [-4.3985975891263305,-2.3223009779757877]},
         {bias,[1.3462974501315348]}],
        [{{1,7.570051345042664e-10},actuator},
         {{0.5,7.570051345042677e-10},neuron},
         {{0,7.570051345042682e-10},neuron}],
        [{{0,7.570051345042682e-10},neuron}]}
{actuator,{{1,7.570051345042659e-10},actuator},
          xor_SendOutput,
          {{origin,7.570051345042693e-10},cortex},
          {private,xor_sim}, 1,
          [{{0.5,7.570051345042677e-10},neuron}],11}
{actuator,{{1,7.570051345042664e-10},actuator},
          xor_SendOutput,
          {{origin,7.570051345042693e-10},cortex},
          {private,xor_sim}, 1,
          [{{0,7.570051345042682e-10},neuron}],
          undefined}
```

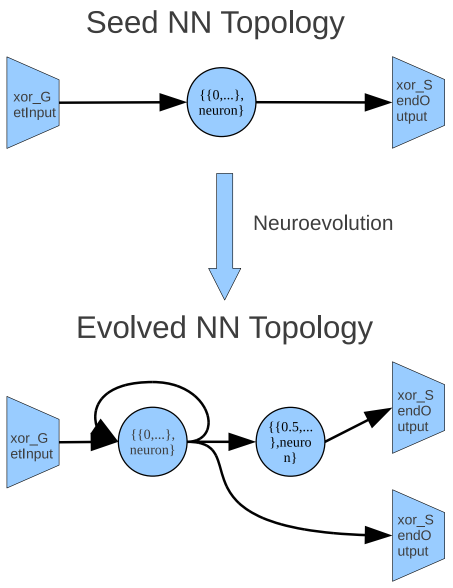


Fig. 9.5 The NN topology of the fittest agent in the population solving the XOR test, from Listing-9.3.

If we ignore the strange part about this NN system having two actuators, the reason behind which we have already solved in Section-9.2.3, we immediately spot another interesting feature. We have evolved a recurrent NN!

A recurrent NN can use memory, which means that the evolved solution, among other things, is most likely also sequence specific. This means that this solution takes into account the order in which the input data is presented. Since in the real world these types of signals would not be presented in any particular order to the XOR logic operator in question, our evolved system would not simulate the XOR operator properly anyway, even after having all the other errors fixed. A proper XOR mimicking neural network must not be sequence specific. Thus it is essential that for this problem we evolve a non recurrent NN system.

We need to be able to control and choose whether we want the evolving neural network systems to have recurrent connections or not. In the same way that we can choose what activation functions the NN system has access to (through the constraint record), we can also specify whether recurrent connections are allowed or not. To add this feature before we can retest our system, we need to: 1. Modify the records.hrl file to add the new element to the constraint tuple. And 2. Modify the genome_mutator module so that it checks whether recurrent or only feedforward connections are allowed, before choosing which elements to link together.

Modifying the records.hrl file is easy, we simply change the constraint record from:

```
-record(constraint,{
    morphology=xor_mimic, %xor_mimic
    neural_afs=[tanh,cos,gaussian,absolute] %[tanh,cos,gaussian,absolute,sin,sqrt,sigmoid]
    }).
```

To:

```
-record(constraint,{
    morphology=xor_mimic, %xor_mimic
    connection_architecture = recurrent, %recurrent|feedforward
    neural_afs=[tanh,cos,gaussian,absolute] %[tanh,cos,gaussian,absolute,sin,sqrt,sigmoid]
    }).
```

The new parameter: *connection_architecture*, can take on two values, either the atom: *recurrent,* or the atom: *feedforward*. Though we've added the new element, *connection_architecture,* to the constraint record, we still need to modify the genome_mutator module so that it actually knows how to use this new parameter. In the genome_mutator module we need to modify all the mutation_operators that add new connections, and ensure that before a new connection is created, the function takes the value of the connection_architecture parameter into consideration. The mutation operators that we need to modify for this are: *add_outlink/1*, *add_inlink/1*, and *add_neuron/1*.

The updated add_outlink/1 function first builds an output id pool, which is a list of all available ids to which the selected neuron can choose to establish a link to. The general id pool is composed by combining together the list of actuator and neuron ids. We must remove from this id list the neuron's own Output_Ids list, which leaves a list of element Ids to which the neuron is not yet connected to. We then check whether the agent allows for recurrent connections, or only feedforward. If recurrent connections are allowed, then a random Id from this list is chosen, and the neuron and the chosen element are linked together. If on the other hand only the feedforward connections are allowed, the neuron's own layer index is checked, and then the composed id pool is filtered such that the remaining id list contains only the element ids whose layer index is greater than that of the neuron. This effectively creates a list of element ids which are 1 or more neural-layers ahead of the chosen neuron, and to whom if a connection is established, would be considered feedforward. To implement this new approach, we convert the original add_outlinke/1 function from:

```
add_outlink(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,
    A_Ids = Cx#cortex.actuator_ids,
    N_Id = lists:nth(random:uniform(length(N_Ids)),N_Ids),
    N = genotype:read({neuron,N_Id}),
    Output_Ids = N#neuron.output_ids,
    case lists:append(A_Ids,N_Ids) -- Output_Ids of
            [] ->
                    exit("********ERROR:add_outlink:: Neuron already connected to all ids");
            Available_Ids ->
                    To_Id = lists:nth(random:uniform(length(Available_Ids)),Available_Ids),
                    link_FromElementToElement(Agent_Id,N_Id,To_Id),
                    EvoHist = A#agent.evo_hist,
                    U_EvoHist = [{add_outlink,N_Id,To_Id}|EvoHist],
                    U_A = A#agent{evo_hist=U_EvoHist},
                    genotype:write(U_A)
    end.
```

To one that uses a filtered neuron id pool, *Outlink_NIdPool,* for the feedforward connections, and the entire id pool for when recurrent connections are allowed:

```
add_outlink(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,
    A_Ids = Cx#cortex.actuator_ids,
    N_Id = lists:nth(random:uniform(length(N_Ids)),N_Ids),
    N = genotype:read({neuron,N_Id}),
    Output_Ids = N#neuron.output_ids,
    Outlink_NIdPool = filter_OutlinkIdPool(A#agent.constraint,N_Id,N_Ids),
    case lists:append(A_Ids,Outlink_NIdPool) -- Output_Ids of
            [] ->
                    exit("********ERROR:add_outlink:: Neuron already connected to all ids");
            Available_Ids ->
                    To_Id = lists:nth(random:uniform(length(Available_Ids)),Available_Ids),
                    link_FromElementToElement(Agent_Id,N_Id,To_Id),
                    EvoHist = A#agent.evo_hist,
                    U_EvoHist = [{add_outlink,N_Id,To_Id}|EvoHist],
                    U_A = A#agent{evo_hist=U_EvoHist},
                    genotype:write(U_A)
    end.
```

The *filter_OutlinkIdPool(Constraint,N_Id,N_Ids)* function has to filter the neuron ids (N_Ids) based on the specification in the constraint record. This new filter_OutlinkIdPool/3 function, is shown in the following listing:

Listing-9.8 The implementation of filter_OutlinkIdPool/3, a constraint based neuron id filtering function.

```
filter_OutlinkIdPool(C,N_Id,N_Ids,Type)->
    case C#constraint.connection_architecture of
            recurrent ->
                    N_Ids;
            feedforward ->
                    {{LI,_},neuron} = N_Id,
                    case Type of
                            outlink ->
                                    [{{Outlink_LI,Outlink_UniqueId},neuron} || {{Outlink_LI,
Outlink_UniqueId}, neuron} <- N_Ids, Outlink_LI > LI];
                            inlink ->
                                    [{{Inlink_LI,Inlink_UniqueId},neuron} || {{Inlink_LI,
Inlink_UniqueId},neuron} <- N_Ids, Inlink_LI < LI]
                    end
    end.
%The function filter_OutlinkIdPool/3 uses the connection_architecture specification in the constraint record of the agent to return a filtered neuron id pool. For the feedforward connection_architecture, the function ensures that only the neurons in the forward facing layers are allowed in the id pool.
```

We can modify the add_inlink/1 mutation operator in the same way. In this function though, if we are to only have feedforward connections, then the filtered neuron id pool needs to have neurons whose layer is less than that of the chosen neuron which is trying to add an inlink. The add_inlink/1 function is modified in the same manner as the add_outlink/1, only we create and use the *filter_InlinkIdPool/3* function instead, which is shown in the following listing:

Listing-9.9 The implementation of filter_InlinkIdPool/3, a constraint based neuron ids filtering function.

```
   filter_InlinkIdPool(C,N_Id,N_Ids)->
         case C#constraint.connection_architecture of
               recurrent ->
                     N_Ids;
               feedforward ->
                     {{LI,_},neuron} = N_Id,
                     [{{Inlink_LI,Inlink_UniqueId},neuron} || {{Inlink_LI,
Inlink_UniqueId},neuron} <- N_Ids, Inlink_LI < LI]
         end.
%The function filter_InlinkIdPool/3 uses the connection_architecture specification in the constraint record of the agent to return a filtered neuron id pool. For the feedforward connection_architecture, the function ensures that only the neurons in the previous layers are allowed in the filtered neuron id pool.
```

Finally, we modify the add_neuron/1 mutation operator. In this operator a new neuron B is created, and is then connected *from* a randomly chosen neuron A, and *to* a randomly chosen neuron C. As in the previous two mutation operators, we compose an Id pool specified by the architecture_constraint parameter, from which the Ids of A and C are then chosen. The modified version of the add_neuron/1 function is shown in Listing-9.10.

Listing-9.10 The modified add_neuron/1 mutation operator, which now uses id pools that satisfy the connection_architecture constraint specification. The bold parts of the code are the added and modified parts of the function.

```
add_neuron(Agent_Id)->
   A = genotype:read({agent,Agent_Id}),
   Generation = A#agent.generation,
   Pattern = A#agent.pattern,
   Cx_Id = A#agent.cx_id,
   Cx = genotype:read({cortex,Cx_Id}),
   N_Ids = Cx#cortex.neuron_ids,
   S_Ids = Cx#cortex.sensor_ids,
   A_Ids = Cx#cortex.actuator_ids,
   {TargetLayer,TargetNeuron_Ids} = lists:nth(random:uniform(length(Pattern)),Pattern),
   NewN_Id = {{TargetLayer,genotype:generate_UniqueId()},neuron},
   U_N_Ids = [NewN_Id|N_Ids],
   U_Pattern = lists:keyreplace(TargetLayer, 1, Pattern,
{TargetLayer,[NewN_Id|TargetNeuron_Ids]}),
   SpecCon = A#agent.constraint,
   genotype:construct_Neuron(Cx_Id,Generation,SpecCon,NewN_Id,[],[]),
   Inlink_NIdPool = filter_InlinkIdPool(A#agent.constraint,NewN_Id,N_Ids),
   Outlink_NIdPool = filter_OutlinkIdPool(A#agent.constraint,NewN_Id,N_Ids),
   FromElementId_Pool = Inlink_NIdPool++S_Ids,
   ToElementId_Pool = Outlink_NIdPool,
   case (Inlink_NIdPool == []) or (Outlink_NIdPool == []) of
       true ->
           exit("********ERROR::add_neuron(Agent_Id)::Can't add new neuron
here, Inlink_NIdPool or Outlink_NIdPool is empty.");
       false ->
           From_ElementId =
lists:nth(random:uniform(length(FromElementId_Pool)),FromElementId_Pool),
           To_ElementId =
lists:nth(random:uniform(length(ToElementId_Pool)),ToElementId_Pool),
   link_FromElementToElement(Agent_Id,From_ElementId,NewN_Id),
   link_FromElementToElement(Agent_Id,NewN_Id,To_ElementId),
       U_EvoHist = [{add_neuron,From_ElementId,NewN_Id, To_ElementId} |
A#agent.evo_hist],
           genotype:write(Cx#cortex{neuron_ids = U_N_Ids}),
           genotype:write(A#agent{pattern=U_Pattern,evo_hist=U_EvoHist})
   end.
```

We do not need to modify outsplice/1 mutation operator, even though it does establish new connections. The reason for this is that if the connection_architecture allows recurrent connections, then there is nothing to modify, and if it is feedforward, then all the connections are already made in the right direction, since if we add a new neuron, we either create a new layer for it, or put it in the layer located between the two spliced neurons, which allows the NN to retain the feedforward structure.

9.3 Retesting Our Neuroevolutionary System

Having now modified all the broken mutation operators, and fixed all the errors, we can compile all the modified modules, and retest our neuroevolutionary system. First, we will once again apply multiple mutation operators to our NN system, and then analyze the resulting NN architecture, manually checking if everything looks as it supposed to. We will then run multiple xor_mimic tests, each test with a slightly different parameter set. This will give us a better understanding of how our system performs.

During this test, we still let the NN evolve recurrent connections. In the following listing we first compile and load the modules by executing *polis:sync()*.We then execute *genome_mutator:long_test(10)*. And then finally, we print the resulting NN system's genotype to console, so that we can visually inspect it:

Listing-9.11 The long_test function applied to our now fixed neuroevolutionary system.

```
3> genome_mutator:long_test(10).
...
4> genotype:print(test).
{agent,test,10,undefined,test, ...
    [{mutate_weights,{{0.5,7.565644036503407e-10},neuron}},
     {add_neuron,{{0.5,7.565644036503407e-10},neuron},
            {{0.5,7.565644036354212e-10},neuron},
            {{0.5,7.565644036503407e-10},neuron}},
     {add_bias,{{0,7.565644036525425e-10},neuron}},
     {add_outlink,{{0.5,7.565644036503407e-10},neuron},
             {{0,7.565644036562396e-10},neuron}},
     {add_outlink,{{0,7.565644036562396e-10},neuron},
             {{0,7.565644036525425e-10},neuron}},
     {mutate_af,{{0,7.565644036535494e-10},neuron}},
     {mutate_af,{{0,7.565644036562396e-10},neuron}},
     {add_bias,{{0,7.565644036535494e-10},neuron}},
     {outsplice,{{0,7.565644036562396e-10},neuron},
           {{0.5,7.565644036503407e-10},neuron},
           {{1,7.565644036562401e-10},actuator}},
     {mutate_af,{{0,7.565644036535494e-10},neuron}},
     {mutate_weights,{{0,7.565644036562396e-10},neuron}},
     {add_inlink,{{0,7.565644036525425e-10},neuron},
            {{0,7.565644036525425e-10},neuron}},
     {add_neuron,{{-1,7.565644036562414e-10},sensor},
            {{0,7.565644036525425e-10},neuron},
            {{0,7.565644036562396e-10},neuron}},
     {add_neuron,{{0,7.565644036562396e-10},neuron},
            {{0,7.565644036535494e-10},neuron},
            {{0,7.565644036562396e-10},neuron}},
     {add_outlink,{{0,7.565644036562396e-10},neuron},
             {{0,7.565644036562396e-10},neuron}}],
    0.13228659163157622,0,
    [{0,
     [{{0,7.565644036525425e-10},neuron},
      {{0,7.565644036535494e-10},neuron},
    {{0,7.565644036562396e-10},neuron}]},
   {0.5,
    [{{0.5,7.565644036354212e-10},neuron},
     {{0.5,7.565644036503407e-10},neuron}]}]}
{cortex,{{origin,7.56564403656243e-10},cortex},
   test,
   [{{0.5,7.565644036354212e-10},neuron},
    {{0.5,7.565644036503407e-10},neuron},
    {{0,7.565644036525425e-10},neuron},
    {{0,7.565644036535494e-10},neuron},
    {{0,7.565644036562396e-10},neuron}],
   [{{-1,7.565644036562414e-10},sensor}],
   [{{1,7.565644036562401e-10},actuator}]}
{sensor,{{-1,7.565644036562414e-10},sensor},
   xor_GetInput,
   {{origin,7.56564403656243e-10},cortex},
   {private,xor_sim},
   2,
   [{{0,7.565644036525425e-10},neuron},
    {{0,7.565644036562396e-10},neuron}],
   3}
{neuron,{{0.5,7.565644036354212e-10},neuron},
   10,
   {{origin,7.56564403656243e-10},cortex},
   absolute,
   [{{{0.5,7.565644036503407e-10},neuron},[-0.07865790723708455]}],
   [{{0.5,7.565644036503407e-10},neuron}],
   [{{0.5,7.565644036503407e-10},neuron}]}
{neuron,{{0.5,7.565644036503407e-10},neuron},
   10,
   {{origin,7.56564403656243e-10},cortex},
   gaussian,
   [{{{0.5,7.565644036354212e-10},neuron},[0.028673644861684]},
    {{{0,7.565644036562396e-10},neuron},[0.344474633962796]}],
   [{{0.5,7.565644036354212e-10},neuron},
    {{0,7.565644036562396e-10},neuron},
    {{1,7.565644036562401e-10},actuator}],
   [{{0.5,7.565644036354212e-10},neuron},
    {{0,7.565644036562396e-10},neuron}]}
{neuron,{{0,7.565644036525425e-10},neuron},
   9,
   {{origin,7.56564403656243e-10},cortex},
   cos,
   [{{{0,7.565644036562396e-10},neuron},[0.22630117969617192]},
    {{{0,7.565644036525425e-10},neuron},[0.06839553053285097]},
    {{{-1,7.565644036562414e-10},sensor},
     [0.4907662278024556,-0.3163769342514735]},
    {bias,[-0.4041650818621978]}],
   [{{0,7.565644036525425e-10},neuron},
    {{0,7.565644036562396e-10},neuron}],
   [{{0,7.565644036525425e-10},neuron},
    {{0,7.565644036562396e-10},neuron}]}
{neuron,{{0,7.565644036535494e-10},neuron},
   7,
   {{origin,7.56564403656243e-10},cortex},
   cos,
   [{{{0,7.565644036562396e-10},neuron},[0.30082326020002736]},
    {bias,[0.00990196169812485]}],
   [{{0,7.565644036562396e-10},neuron}],
   [{{0,7.565644036562396e-10},neuron}]}
{neuron,{{0,7.565644036562396e-10},neuron},
   9,
   {{origin,7.56564403656243e-10},cortex},
   tanh,
   [{{{0.5,7.565644036503407e-10},neuron},[0.29044390963714084]},
    {{{0,7.565644036525425e-10},neuron},[-0.11820697604732322]},
    {{{0,7.565644036535494e-10},neuron},[2.203261827127093]},
    {{{0,7.565644036562396e-10},neuron},[0.13355748834368064]},
    {{{-1,7.565644036562414e-10},sensor},
     [-2.786539611443157,3.0562965644493305]}],
   [{{0,7.565644036525425e-10},neuron},
    {{0.5,7.565644036503407e-10},neuron},
    {{0,7.565644036535494e-10},neuron},
    {{0,7.565644036562396e-10},neuron}],
   [{{0,7.565644036525425e-10},neuron},
    {{0,7.565644036535494e-10},neuron},
    {{0,7.565644036562396e-10},neuron}]}
{actuator,{{1,7.565644036562401e-10},actuator},
    xor_SendOutput,
     {{origin,7.56564403656243e-10},cortex},
     {private,xor_sim},
     1,
     [{{0.5,7.565644036503407e-10},neuron}],
     5}
```

It works! Figure-9.6 shows the visual representation of this NN system's topology. If we inspect the mutation operators, and the actual connections, everything is in perfect order.

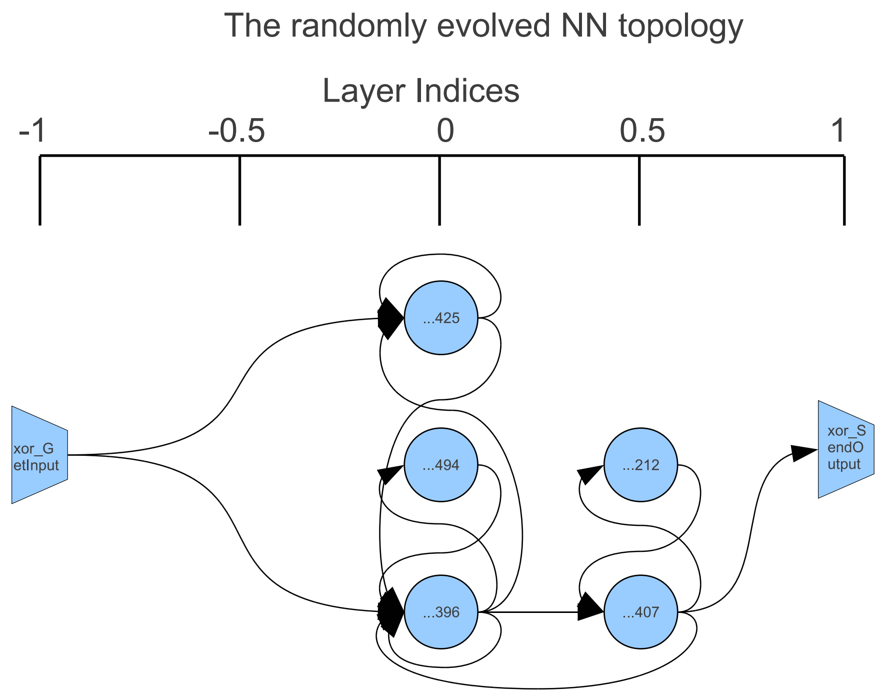


Fig. 9.6 The randomly evolved topology through the genome_mutator:long_test(10) execution.

We will now test our system on the xor_mimic problem with the following set of parameters:

1. Constraint's activation functions set to [tanh], and MAX_ATTEMPTS to 50, 10, and 1:
   This is done by changing the MAX_ATTEMPTS in the exoself module, for each separate test.
2. Activation functions are not constrained, *connection_architecture* is set to *feedforward*, and MAX_ATTEMPTS is set to 50, 10, and 1:
   This is done by changing the INIT_CONSTRAINTS in the population_monitor module from one which previously constrained the activation functions, to one that no longer does so:

```
-define(INIT_CONSTRAINTS,[#constraint{morphology=Morphology,neural_afs=
Neural_AFs, connection_architecture=CA} || Morphology<-[xor_mimic],Neural_AFs<-
[[tanh]], CA<-[feedforward]]).
```

To:

```
-define(INIT_CONSTRAINTS, [#constraint{morphology=Morphology,
connection_architecture=CA} || Morphology<-[xor_mimic], CA<-[feedforward]]).
```

We have developed different kinds of activation functions, and created our neuroevolutionary system to give NN systems the ability to incorporate these various functions based on their need. Also, the MAX_ATTEMPTS variable specifies the duration of the tuning phases, how well each topology is tested before it is given its final fitness score. A neuroevolutionary setup using MAX_ATTEMPTS = 1 is equivalent to it using a standard genetic algorithm rather than a memetic algorithm based approach, since the tuning phase then only acts as a way to assess the NN system's fitness, and all the mutation operators (including the weight perturbation) are applied in the topological mutation phase. When the MAX_ATTEMPTS variable is set to 50, then each topology is tuned for a considerable amount of time.

To acquire the test-results of the above specified setup, we first set the parameters: INIT_CONSTRAINTS and the MAX_ATTEMPTS, to their new values, then run polis:sync() to update and load the modified modules, and then run the population_monitor:test() function to perform the actual test, the results of which are shown next:

Activation function: tanh, MAX_ATTEMPTS=50:

```
******** Population_Monitor:test shut down with Reason:normal OpTag:continue, while in
OpMode:gt
******** Tot Agents:10 Population Generation:25 Eval_Acc:14806 Cycle_Acc:59224
Time_Acc:8038997
```

With the last generation's NN systems having the number of neurons ranging from: 6-9.

Activation function: tanh, MAX_ATTEMPTS=10:

```
******** Population_Monitor:test shut down with Reason:normal OpTag:continue, while in
OpMode:gt
******** Tot Agents:10 Population Generation:33 Eval_Acc:5396 Cycle_Acc:21584
Time_Acc:2456883
```

With the last generation's NN systems having the number of neurons ranging from: 7-9.

Activation function: tanh, MAX_ATTEMPTS=1:

```
******** Population_Monitor:test shut down with Reason:normal OpTag:continue, while in
OpMode:gt
******** Tot Agents:11 Population Generation:100 Eval_Acc:2281 Cycle_Acc:9124
Time_Acc:2630457
```

In this setup, the system failed to produce a solution, with the maximum fitness reached being ~7. This is understandable, since in the standard genetic algorithm's 97% of the mutations are weight perturbation based mutations, with the remainder being topological mutation operators. In our setup though, because our system does weight tuning in a different phase, the topological mutation phase uses the weight_perturbation operator with the same probability as any other. We will change this in the future.

Activation function: tanh, cos, gaussian, absolute MAX_ATTEMPTS=50:

```
******** Population_Monitor:test shut down with Reason:normal OpTag:continue, while in
OpMode:gt
******** Tot Agents:9 Population Generation:1 Eval_Acc:910 Cycle_Acc:3640
Time_Acc:234083
```

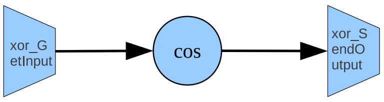
Activation function: tanh, cos, gaussian, absolute MAX_ATTEMPTS=10:

```
******** Population_Monitor:test shut down with Reason:normal OpTag:continue, while in
OpMode:gt
******** Tot Agents:10 Population Generation:4 Eval_Acc:694 Cycle_Acc:2776
Time_Acc:209243
```

Activation function: tanh, cos, gaussian, absolute MAX_ATTEMPTS=1:

```
******** Population_Monitor:test shut down with Reason:normal OpTag:continue, while in
OpMode:gt
******** Tot Agents:9 Population Generation:22 Eval_Acc:565 Cycle_Acc:2260
Time_Acc:266885
```

```
{agent,{7.565639675302518e-10,agent},
     0,undefined,7.565639675302535e-10,
     {{origin,7.565639675302501e-10},cortex},
     {[{0,1}],[],[{sensor,undefined,xor_GetInput,undefined,
{private,xor_sim},2,[],0}],
      [{actuator,undefined,xor_SendOutput,undefined,
       {private,xor_sim},1,[],0}]},
       {constraint,xor_mimic,feedforward,
       [tanh,cos,gaussian,absolute]},[],4071.089031109478,0,
       [{0,[{{0,7.565639675302466e-10},neuron}]}]}
{cortex,{{origin,7.565639675302501e-10},cortex},
      {7.565639675302518e-10,agent},
      [{{0,7.565639675302466e-10},neuron}],
      [{{-1,7.565639675302489e-10},sensor}],
      [{{1,7.565639675302477e-10},actuator}]}
{sensor,{{-1,7.565639675302489e-10},sensor},
      xor_GetInput,{{origin,7.565639675302501e-10},cortex},
      {private,xor_sim},2,
      [{{0,7.565639675302466e-10},neuron}],0}
{neuron,{{0,7.565639675302466e-10},neuron},
      0,{{origin,7.565639675302501e-10},cortex},
      cos,[{{{-1,7.565639675302489e-10},sensor},
      [-4.640518254468062,-4.789381628869486]}],
      [{{1,7.565639675302477e-10},actuator}],[]}
{actuator,{{1,7.565639675302477e-10},actuator},
      xor_SendOutput,{{origin,7.565639675302501e-10},cortex},
      {private,xor_sim},1,[{{0,7.565639675302466e-10},neuron}],0}
```



The evolved solution
W1: -4.640518
W2: -4.789381

Input: [-1,-1]
Output: cos(-4.64*-1 + -4.79*-1) = -0.9999

Input: [-1, 1]
Output: cos(-4.64*-1 + -4.79*1) = 0.9889

Input: [1,-1]
Output: cos(-4.64*1 + -4.79*-1) = 0.9889

Input: [1, 1]
Output: cos(-4.64*1 + -4.79*1) = -0.9999

Fig. 9.7 The discovered solution for the XOR problem, using only a single neuron.

The benchmark results when we allow for all activation functions to be used, are remarkably different. We've developed our neuroevolutionary system to allow the evolving NN systems to efficiently incorporate any available activation functions. In these last 3 scenarios, the evolved solutions all contained a single neuron, as shown in Fig-9.7. In all 3 tests the solutions were reached within 1000 evaluations, very rapidly. The discovered solution? It was a single neuron without a bias, using a *cos* activation function.

We have now tested our neuroevolutionary system on the basic benchmark problem. We have confirmed that it can evolve solutions, that it can evolve topologies and synaptic weights, that those solutions are correct, and that the evolved topologies are as expected. Though we've only developed a basic neuroevolutionary system thus far, it is decoupled and general enough that we can augment it, and easily improve it further, which is exactly what we will do in later chapters.

9.4 Summary

In this chapter we have thoroughly tested every mutation operator that we've added in the previous chapter. Though initially the mutation operator tests seemed successful, when testing our system on the XOR problem, and applying numerous mutation operators and then analyzing the evolved topology manually, we noticed errors to be present. We explored the origin of these detected errors, and then corrected them, re-testing our system on the XOR problem, successfully so.

The evolutionary algorithms built to evolve around problems, will also result in being able to evolve around small errors present in the algorithm itself. Thus, though it may seem that a test ran to completion, and did so successfully, as we've found out in this chapter, sometimes it is worthwhile to analyze the results, and the evolved agents, manually. It is during the thorough manual analysis that the more difficult to find errors are discovered. We have done just that in this chapter, and gained experience in the process of performing manual analysis of evolved NNs. This will give us an advantage in the future, as we continue adding more advanced features to our system, which will require debugging sooner or later.