Chapter 7 Adding the "Stochastic Hill-Climber" Learning Algorithm

Abstract In this chapter we discuss the functionality of an optimization method called the *Stochastic Hill Climber*, and the *Stochastic Hill Climber With Random Restarts*. We then implement this optimization algorithm, allowing the exoself process to train and optimize the neural network it is overlooking. Afterwards, we implement a new problem interfacing method through the use of public and private *scapes*, which are simulated environments, not necessarily physical. We apply the new system to the XOR emulation problem, testing its performance on it. Finally, looking to the future and the need for us to be able to test and benchmark our neuroevolutionary system as we add new features to it, we create the *benchmarker* process, which summons the trainer and the NN it trains, applying it to some specified problem X number of times. Once the benchmarker has applied the trainer to the problem X number of times and accumulated the resulting statistics, it calculates the averages and the associated standard deviations for the important performance parameters of the benchmark.

Though we have now created a functional NN system, synchronized by the cortex element and able to use sensors and actuators to interact with the world, it still lacks the functionality to learn or be trained to solve a given problem or perform some given task. What our system needs now is a learning algorithm, a method by which we can train the NN to do something useful. Beside the learning algorithm itself, we will also need some external system that can automatically train and apply this learning algorithm to the NN, and a system that can monitor the NN for any unexpected crashes or errors, restoring it to a functional state when necessary. Finally, though we have implemented sensors and actuators which the NN can use to interact with the world and other programs, we have not yet developed a way to present to the NN the problems we wish it to solve, or tasks that we wish to train it to perform. It is these extensions that we will concern ourselves with in this chapter.

Here we will continue extending the NN system we've developed in the previous chapter. In the following sections we will discuss and add these new algorithms and features, and develop the necessary modules and module extensions to incorporate the new functionality.

7.1 The Learning Method

An evolutionary algorithm (EA) is a population based optimization algorithm which works by utilizing biologically analogous operators: Selection, Mutation, and Crossover. We will begin developing a population based approach in the next chapter, at this point though, we will still use a single NN based system. In such a case, there is one particular optimization algorithm that is commonly compared to an EA due to some similarities, that algorithm is the Stochastic Hill-Climber (SHC), and is the one we will be implementing.

In an evolutionary algorithm we use a population of organisms/agents, all of whom we apply to the same problem in parallel. Afterwards, based on the fitness each agent demonstrated, we use a selection algorithm to pick the fit from the unfit, and then create offspring from these chosen fit agents. The offspring are created by taking the fit agents and perturbing/mutating them, or by crossing two or more fit agents together to create a new one. The new population is then composed from some combination of the original fit agents and their offspring (their mutated forms, and/or their crossover based combinations). In this manner, through selection and mutation, with every new generation we produce organisms of greater and greater fitness.

In the Stochastic Hill-Climbing algorithm that we will apply to a single agent, we do something similar. We apply a single NN to a problem, gage its performance on the problem, save its genotype and fitness, and then mutate/perturb its genome in some manner and then re-apply the resulting mutant to the problem again. If the mutated agent performs better, then we save its genotype and fitness, and mutate it again. If the mutated agent performs worse, then we simply reset the agent’s genotype to its previous state, and mutate it again to see if the new mutant will perform better. In this manner we slowly climb upwards on the fitness landscape, taking a step back if the new mutant performs worse, and retrying in a different direction until we generate a mutant that performs better. If at some point it is noticed that no new mutants of this agent (in our case the agent is a NN based system) seem to be increasing in fitness, we then consider our agent to have reached a local optimum, at which point we could save its genotype and fitness score, and then apply this fit NN to the problem it was optimized for. Or we could try to restart the whole thing again, generate a completely new random genotype and try to hill-climb it to greater fitness. By generating a completely new NN genotype we hope that its initial weight (or topology and weight) combination will spawn it in a new and perhaps better location of the fitness landscape, from which a higher local optimum is reachable. In this manner the random restart stochastic hill-climber optimization algorithm can reach local and even global optimum.

The following steps represent the Stochastic Hill-Climbing (SHC) algorithm:

```
1. Repeat:
    2. Apply the NN to some problem.
    3. Save the NN's genotype and its fitness.
    4. Perturb the NN's synaptic weights, and re-apply the NN to the same problem.
    5. If the fitness of the perturbed NN is higher, discard original NN and keep
       the new. If the fitness of the original NN is higher, discard the perturbed
       NN and keep the old.
6. Until: Acceptable solution is found, or some stopping condition is reached.
7. Return: The genotype with the fittest combination of weights.
```

The algorithm is further extended by generating completely new random genotypes when the NN currently being trained ceases to make progress. The following steps extend the SHC to the Random-Restart SHC version:

```
8. Repeat:
    9. Generate a completely new genotype, and perform steps 1-7 again.
    10.If the new genotype reaches a higher fitness, overwrite the old genotype
       with the new one. If the new genotype reaches a lower fitness, discard the
       new genotype.
11.Until: Acceptable solution is found, or some stopping condition is reached.
12.Return: The final resulting genotype with its fitness score.
```

Steps 8-11 are necessary in the case that the original/seed combination of NN topology and weights could not be optimized (hill climbed) to the level that would solve the problem. Generating a new genotype, either of the same topology but with different random weights, or of a different topology and weights, could land this new genotype in a place of the fitness landscape from which a much higher fitness can be reached. We will create a process which can apply steps *8-12* to a NN system, and we'll call this process: **trainer**.

We will not use the most basic version of the SHC. Instead we will modify it to be a bit more flexible and dynamic. Rather than perturbing the NN's weights some predefined *K* number of times before giving up, we will make *K* dynamic and based on a *Max_Attempts* value, which itself can be defined by the researcher or based on some feature of the NN that is being optimized. If for example we have perturbed the NN's weights *K==Max_Attempts* number of times, and none of them improved the NN's fitness, this implies that based on where it was originally created on the fitness landscape, the particular genotype has reached a good combination of weights, and that the NN is roughly the best that its topology allows it to be. But if on *K =< Max_Attempts* the NN's fitness improves, we reset K back to 0. Since the NN has just improved its fitness, it might have just escaped from some local optimum on the fitness landscape, and thus deserves another full set of attempts to try and improve its weights further. Thus, before the NN is considered to be at the top of its reachable potential, it has to fail to improve its fitness **Max_Attempts** number of times *in a row*. Max_Attempts can be set up by the researcher, the larger the Max_Attempts value, the more processing we're willing to spend on tunning the NN's synaptic weights. This method will ensure that if the NN can jump to a fitter place from the location on the fitness landscape that it's currently on, it will be given that chance. Using this augmented SHC algorithm we will tune the NN's synaptic weights, allowing the particular NN topology to reach its potential, before considering that this NN and its topology is at its limits.

The number of times the trainer process should create a new genotype (same topology but with a new set of random weights), will also be based on a similar approach. The trainer process will use *Trainer_MaxAttempts* variable for this. And it is only after Trainer_MaxAttempts number of genotypes in a row fail to produce higher fitness, will the training phase be considered complete. Once training is complete, the trainer process will return the best found genotype, and store it in the file with the name "best". This augmented SHC is diagrammed in Fig-7.1.

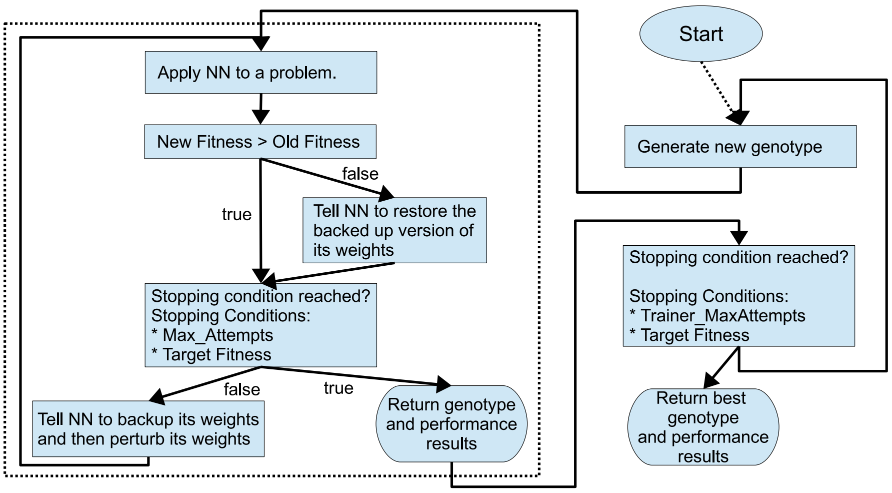
Start
Generate new genotype
Apply NN to a problem.
New Fitness > Old Fitness
false
true
Tell NN to restore the backed up version of its weights
Stopping condition reached? Stopping Conditions: * Max_Attempts * Target Fitness
false
true
Tell NN to backup its weights and then perturb its weights
Return genotype and performance results
Stopping condition reached?
Stopping Conditions: * Trainer_MaxAttempts * Target Fitness
Return best genotype and performance results

Fig. 7.1 Augmented Stochastic Hill Climber algorithm.

Further deviation from the standard algorithm will be with regards to the intensity of the weight perturbations we'll apply, and which weights and neurons we will apply those perturbations to:

1. Each neuron in the trained NN will be chosen for perturbation with a probability of 1/sqrt(NN_Size), where NN_Size is the total number of neurons in the NN.
2. Within the chosen neuron, the weights which will be perturbed will be chosen with the probability of 1/sqrt(Tot_Weights), where Tot_Weights is the total number of synaptic weights in the weights list.
3. The intensity of the synaptic weight perturbation will be randomly chosen with uniform distribution between -Pi and Pi.
4. Max_Attempts variable will be proportional to sqrt(NN_Size).

Features 1, 2 & 3 allow our system to have a chance of producing very high intensity mutations, where perturbations are large and applied to many neurons and many synaptic weights within those neurons, all at once. At the same time there is a chance that the general perturbation intensity will be small, and applied only to a few neurons and a few of their synaptic weights. Thus this approach allows our system to use highly variable mutation/perturbation intensities. Small intensity mutations can fine tune the weights of the NN, while large intensity mutations give our NN a chance to jump out of local synaptic weight optima. Feature 4 will ensure that larger NNs will have a greater amount of processing resources allocated to the tuning of their synaptic weights, because the larger the NN the greater the chance that a larger combination of weights needs to be set to just the right values all at once for the NN to become fitter, which requires a larger number of attempts to do so.

7.1.1 Comparing EA to Random Restart SHC

Note how similar the random-restart stochastic hill-climber (RR-SHC) and the standard evolutionary algorithm (EA) are. The RR-SHC is almost like a sequential version of the EA. The following list is a comparison of that:

1.

– **EA**: In generational EA we start by creating a population of NNs, each with the same topology and thus belonging to the same species. Though each individual will have the same NN topology, each one will have a different combination of synaptic weights.

– **RR-SHC**: In RR-SHC we create a particular genotype, a NN topology. We then try out different combinations of weights. Each new combination of weights represents an individual belonging to the same species. We are thus trying out the different individuals belonging to the same specie sequentially, one at a time, instead of all in parallel.
**But unlike in EA, these different individuals belonging to the same species are not generated randomly, and are not tried in a random order. Instead we generate one individual by perturbing its "ancestor". Each new individual in the specie is based on the previous version of itself, which is used as a stepping stone in an attempt to create a fitter version.*

2.

– **EA**: After all the organisms have been given their fitness, we create the next generation from the subset of the fit organisms. The next generation might contain new NN topologies and new species, generated from the old species through mutation and crossover. Again, each NN species will contain multiple members whose only difference is in their synaptic weights.
**But here the new species are generated as an attempt at improvement (hill climbing), with the parent individuals acting as the stepping stone for the next topological and parametric innovation.*

- **RR-SHC**: After trying out the different permutations of synaptic weights for the original topology, we generate a new one (either the same or a new topology). We then again try out the different synaptic weight combinations for the genotype that starts from a different spot on the fitness landscape.
  **But here we are trying out the same topology. Although of course the algorithm can be augmented, and during this step we could generate a new genotype, not simply having a different set of starting weights but one having a topology that is a perturbation of the previous version, or even completely new and random.*

The EA will eventually have a population composed of numerous species, each with multiple members, and thus EA explores the solution space far and wide. If one species or a particular combination of weights in a species leads to a dead end on the fitness landscape, there will be plenty of others working in parallel which will, if there is a path, find a way to higher fitness by another route.

On the other hand the RR-SHC, during its generation of a new genotype or perturbation of weights, can only select that 1 option, the first combination that leads to a greater fitness is the path selected. If that particular selection is on the path to a dead end, it will not be known until it's too late and the system begins to fail to produce individuals of greater fitness. A dead-end that is reached when perturbing weights is not so bad because our system also generates and tries out new genotypes. But if we start generating new topologies based on the old ones in this sequential hill climbing manner, then the topological dead end reached at this point could be much more dire, since we will not know when the first step towards the dead end was taken, and we will not have multiple other species exploring paths around the topological dead end in parallel... Fig-7.2 shows the similarities and differences in the evolutionary paths taken by organisms optimized using these two methods.

In the following figure, the red colored NNs represent the most fit NNs within the particular specie they belong to. For the evolutionary computation, the flow of time is from top to bottom, whereas for the RR-SHC it is from top to bottom for a species, but the NNs and species are tried one after the other, so it is also from left to right. Evolutionary process for both, the evolutionary computation algorithm and the RR-SHC algorithm, is outlined by the step numbers, discussed next.

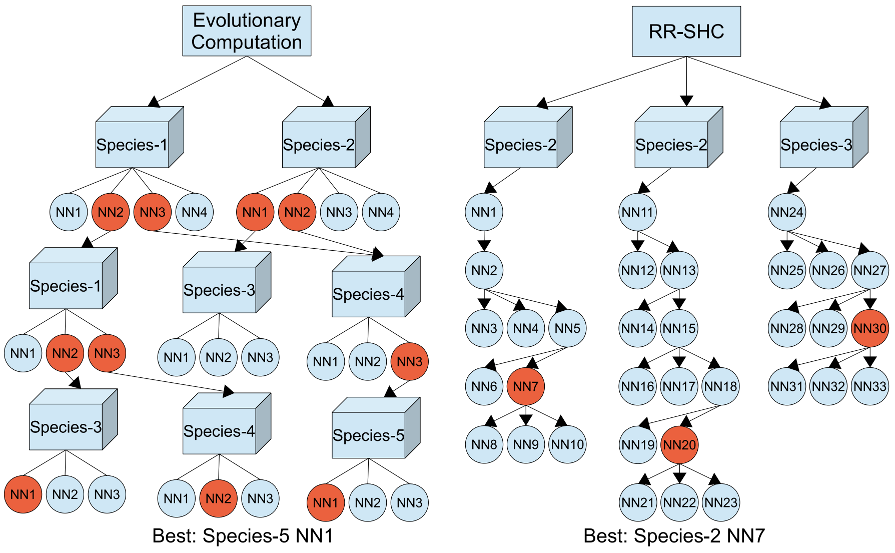

Fig. 7.2 Similarities and Differences in the evolutionary paths taken by organisms evolved through EA and RR-SHC.

Let us first go through the evolutionary path taken by the *Evolutionary Computation* algorithm, by following the specified steps:

1. Species-1 and Species-2 are created, each with 4 NNs, thus the population is composed of 8 NNs in total, separated into 2 species.
2. In Species-1, NN2 and NN3 are the most fit. NN2 creates 3 offspring, which all belong to Species-1. NN3 creates offspring which belong to Species-4. In Species-2, NN1 and NN2 are the most fit. The NN1 of Species-2 creates offspring which make up Species-3, while NN2 creates offspring which go into Species-4. In this scenario, the parents do not survive when creating an offspring, and thus Species-2 does not make it to the next generation.
3. During the second generation, the population is composed of Species-1, Species-4, and Species-3, and 9 NNs in total. In this population, the most fit NNs are NN2 and NN3 of Species-1, and NN3 of Species-4. NN2 of Species-1 creates 3 offspring, all of which have the topology of Species-3. NN3 creates 3 offspring, all of which have the topology of Species-4. Finally, NN3 of Species-4 creates 3 offspring, all of which have a new topology, and are thus designated as Species-5.
4. The third generation, composed of Species-3, Species-4, and Species-5, make up the population of size 9. The most fit of this population is the NN1 of Species-3, NN2 of Species-4, and NN1 of Species-5. The most fit of the three NNs, is NN1 of Species-5, which is designated as Champion.

Similarly, let us go through the steps taken by the RR-SHC algorithm:

1. The RR-SHC creates Species-2 (if we are to compare the topologies to those created by the evolutionary computation algorithm that is, otherwise we can designate the topology with any value).
2. First NN1 is created and tested. Then NN2 is created by perturbing NN1. NN2 has a higher fitness than NN1. The fitness has thus just increased, and NN2's fitness is now the highest reached thus far for this run.
3. We perturb NN2, creating a NN3. NN3 does not have a higher fitness value, so we re-perturb NN2 and create NN4, which also does not have a higher fitness than NN2. We perturb NN2 again and create NN5, which does have a higher fitness than NN2. We designate NN5 as the most fit at this time.
4. NN5 is perturbed to create NN6, which is not fitter. NN5 is perturbed again to create NN7, which is fitter than NN5, thus NN7 is now designated as the most fit.
5. NN7 is perturbed to create NN8, which is not fitter, and it is then perturbed to create NN9, which is also not fitter. We perturb NN7 for the third time to create NN10, which is also not fitter. If we assume that we set our RR-SHC algorithm's Max_Attempts to 3, then our system has just failed to produce a fitter agent 3 times in a row. Thus we designate NN7 of this optimization run, to be the most fit.
6. NN7 is saved as the best NN achieved during this stochastic hill climbing run.
7. The RR-SHC creates a new random NN11, whose topology designates it to be of Species-3.
8. The NN11 is perturbed to create NN12, which is not fitter than NN11. It is perturbed again to create NN13, which is fitter than NN11, and is thus designated as the fittest thus far achieved in this particular optimization run.
9. This continues until termination condition is reached for this hill climbing run as well. At which point we see that NN20 is the most fit.
10. A new random NN21 is generated. It is optimized through the same process. Until finally NN30 is designated to be the most fit of this hill climbing run.
11. Amongst the three most fit NNs, [NN7, NN20, NN30], produced from 3 random restarts of the hill climbing optimization algorithm, NN7 is the most fit. Thus, NN7 is designated as the champion produced by the RR-SHC algorithm.

We will return to these issues again in later chapters. And eventually build a hybrid composed of these two approaches, in an attempt to take advantage of the greedy and effective manner in which SHC can find good combinations of synaptic weights, and the excellent global optima searching abilities of the population based evolutionary approach.

7.2 The Trainer

The trainer will be a very simple program that first generates a NN genotype under the name "experimental", and then applies it to the problem. After the exoself (discussed in the next section) finishes performing synaptic weight tuning of its NN, it sends the trainer process a message with the NN's highest achieved fitness score and total number of evaluations it took to reach it (the number of times NN's total fitness had been evaluated). The trainer will then compare this NN's fitness to the fitness of the genotype under the name "best" (which will be 0 if there is no genotype under such name yet). If the fitness of "experimental" is higher than that of "best", the trainer will rename experimental to best, thus overwriting and removing it. If the fitness of "best" is higher than that of "experimental", the trainer will not overwrite the old genotype. In either case, the trainer then generates a new genotype under the name "experimental", and repeats the process.

As noted in section 7.1, the trainer will use *Trainer_MaxAttempts* variable to determine how many times to generate a new genotype. Only once the Trainer fails to generate a fitter genotype *Trainer_MaxAttempts* number of times in a row, will it be finished, at which point the trainer process will have stored the fittest genotype in the file "best".

7.3 The Exoself

We now need to create an external process to the NN system which tunes the NN's weights through the use of augmented SHC algorithm. A modified version of the *Exoself* process we created in the previous chapter is an excellent contender for this role. We cannot use the *cortex* element for this new functionality because it is part of the NN system itself. The cortex is the synchronizer, and also the element that can perform duties that require global view of the neural topology. For example in something like competitive learning where the neural response intensities to some signal need to be compared to one another for the entire network, an element like cortex can be used. But if for example it is necessary to shut down the entire NN system, or to add new sensors and actuators or new neurons while the NN itself is live and running, or update the system's source code, or recover a previous state of the NN system, that duty could be better performed by an external process like the *exoself.*

Imagine an advanced ALife scenario where a simulated organism is controlled by a NN. In this simulation the organism is already able to modify its own neural topology and add new sensors and actuators to itself. To survive in the environment, it learned to experiment with its own sensors, actuators, and neural architecture so as to give itself an advantage. During one of its experimentations, something goes terribly wrong: synchronization becomes broken and the whole NN system stops functioning. For example the NN based agent (an *infomorph*) is experimenting with its own neural topology, and by mistake deletes a large chunk of itself. The system would become too broken to fix itself after such a thing, and thus the problem would have to be fixed from the outside of the NN system, by some process that is monitoring it and can revert it back to its previous functional state. For such, and more common events, we need a process which would act as a constant *monitor of the self*, *while being external to the self*. This process is the *Exoself*.

The exoself is a process that will function in a cooperative manner with the self (NN). It will perform jobs like backing up the NN’s genotype to file, reverting to earlier versions of the NN, adding new neurons in live systems when asked by the NN itself... In slightly less advanced scenarios, the exoself will be used to optimize the weights of the NN it is monitoring. Since the exoself is outside the NN, it could keep track of which combination of weights in the NN produced a fitter individual, and then tell the NN when to perturb its neural weights, and when to revert to the previous combination of the weights if that yielded a fitter form. The architecture of such a system is presented in Fig-7.3.

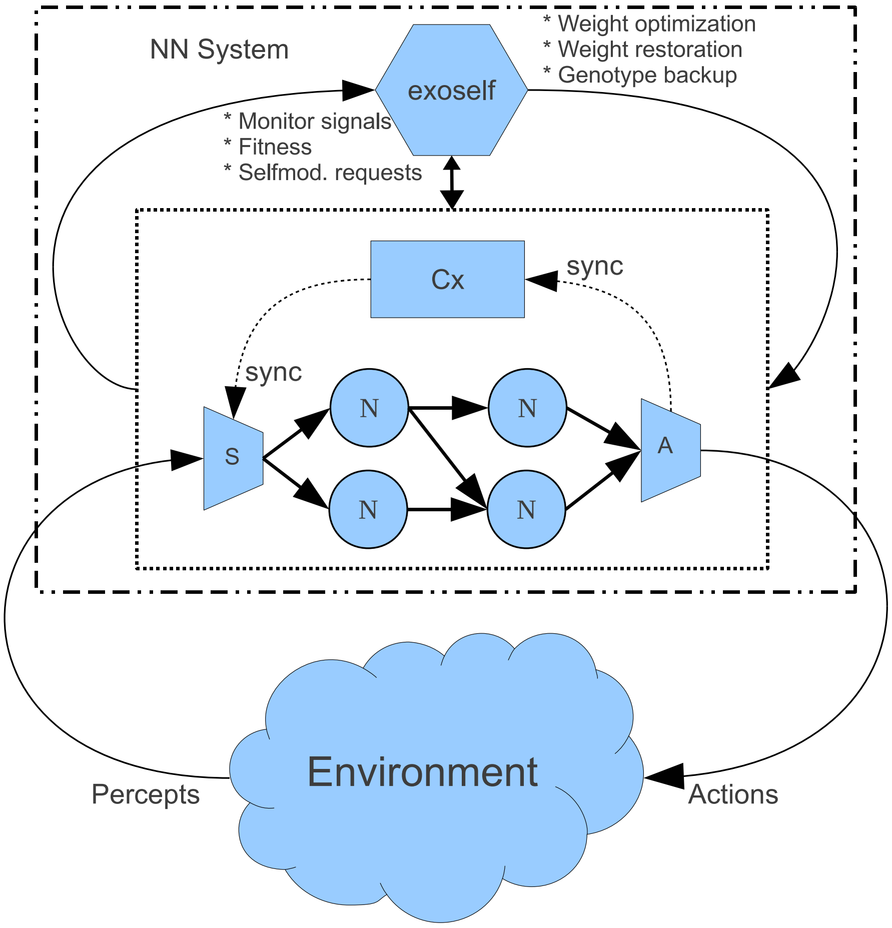

Fig. 7.3 The architecture of a NN system with an exoself process.

Having now covered the trainer and the exoself, which act as the appliers of the augmented RR-SHC optimization algorithm to the NN system, we now need to come up with a way to apply the NN to the problems, or present tasks to the NN that we wish it to learn how to perform. The simulations, tasks, and problems that we wish to apply our NN to, will be abstracted into *scape* packages, which we discuss in the next section.

7.4 The Scape

Scapes are composed of two parts, a simulation of an environment or a problem we are applying the NN to, and a function that can keep track of the NN's performance. Scapes run outside the NN systems, as independent processes with which the NNs interact using their sensors and actuators. There are 2 types of scapes. One type of scapes, private, is spawned for each NN during the NN's creation, and destroyed when that NN is taken offline. Another type of scapes, public, is persistent, they exist regardless of the NNs, and allow multiple NNs to interact with them at the same time, and thus they can allow those NNs to interact with each other too. The following are examples of these two types of scapes:

1. For example, let's assume we wish to use a NN to control a simulated robot in a simulated environment where other simulated robots exist. The fitness in this environment is based on how many simulated plants the robot eats before running out of energy. The robot's energy decreases at some constant rate. First we generate the NN with appropriate Sensors and Actuators with which it can interface with the scape. The sensors could be cameras, and the actuators could control the speed and direction of the robot's navigation through the environment, as shown in Fig-7.4. This scape exists outside the NN system, and for the scape to be able to judge how well the NN is able to control the simulated robot, it needs a way to give the NN fitness points. Furthermore, the scape can either give the NN a fitness point every time the robot eats a plant (event based), or it can keep track of the number of plants eaten throughout the robot's lifetime, until the robot runs out of energy, at which point the scape would use some function to give the NN its total fitness (life based) by processing the total number of plants eaten using that function. The simulated environment could have many robots in it, interacting with the simulated environment and each other. When one of the simulated robots within the scape runs out of energy, or dies, it is removed from the scape, while the remaining organisms in the scape persist. The scape continues to exist independently of the NNs interacting with it, it is a *public scape*.

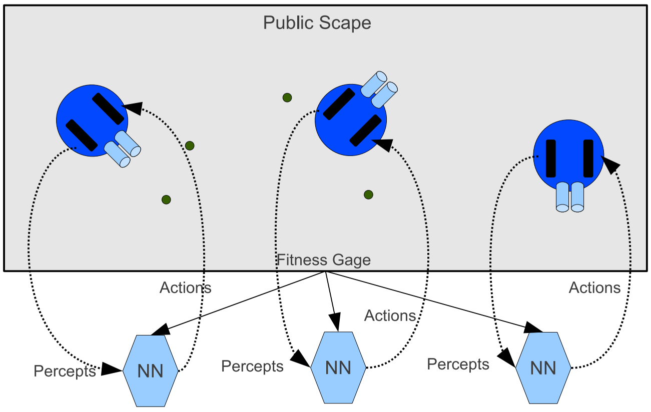

Fig. 7.4 Public simulation, with multiple organisms being controlled by NNs.

2. We want to create a NN that is able to balance a pole on a cart which can be pushed back and forth on a 2 meter track, as shown in Fig-7.5. We could create a scape with the physics simulation of the pole and the cart, which can be interacted with through the NN’s sensors and actuators. The job of the NN would be to push the cart back and forth, thus balancing the pole on it. The NN’s sensors would gather information like the velocity and the position of the cart, and the angular velocity and the position of the pole. The NN’s actuators would push the cart back and forth on the track. The scape, having access to the whole simulation, could then distribute fitness points to the NN based on how well and how long it balances the pole. When the NN is done with its pole balancing training and is deactivated, the scape is deactivated with it. If there are 3 different NNs, each will have its own pole balancing simulation scape created, and those simulations will only exist for as long as the NNs exist. Some problems or simulations are created specifically, and only, for that NN which needs it. This is an example of a *private scape*.

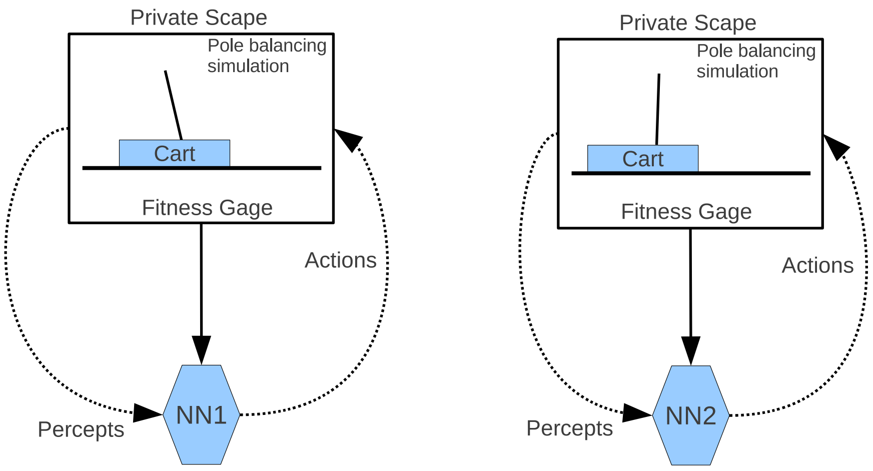

Fig. 7.5 Private simulation, with multiple NN based agents, each with its own private scape.

Though both of these problems (Artificial life and pole balancing) are represented as scapes, they have an important difference. The first scape is a 3d environment where multiple robots can exist, the scape's existence is fully independent of the NNs interacting with it, it is a public scape. The scape exists on its own, whether there are organisms interacting with it or not. It has to be spawned independently and before the NNs can start interacting with it. The second scape example on the other hand is created specifically for each NN when that neural network goes online, it is an example of a private scape. A private scape is in a sense summoned by the NN itself for the purpose of practicing something, or training to perform some task in isolation, a scape to which other organisms should not have access. It's like a privately spawned universe which is extinguished when the NN terminates.

We will present the problems we wish our NNs to learn to solve through these scape environments. Furthermore, we will specify within the sensors and actuators which scapes the NN should spawn (if the scape is private), or interface with (if the scape is public). Sensors and actuators are created to interact with things, thus they provide a perfect place where we can specify what scapes, if any, they should spawn or interface with. The next important design decision is: how do we want the scapes to notify the NNs of their fitness scores. A problem we solve in the next section.

7.5 Scapes, Sensors, Actuators, Morphologies, and Fitness

Now that we've agreed on the definition of a scape, (a simulation of an environment, not necessarily physical or 3 dimensional, that can also gage fitness and which can be interfaced with, through sensors and actuators), we will need to devise a way by which the Scape can contact the NN and give it its fitness score.

The NN uses the sensors and actuators to interface with the scape, and since a scape exists as a separate process, the sensors and actuators will interact with it by sending it messages. When the cortex sends a particular sensor the {Cx_PId, sync} message, that sensor is awoken to action and if it is the type of sensor that interfaces with a scape, it will send the scape a message and tell it what kind of sensory data it needs. We will make the message from sensor to scape have the following format: {Sensor_PId, sensor_name}, and the scape will then send the Sensor_PId the sensory signals based on the sensor_name. If the sensor is a camera, it will tell the scape that it needs camera based data associated with that sensor. If the sensor is a sonar scanner, then it would request sonar data. The scape then sends some *sensor-specific* data as a reply to the sensor's message. Finally, the sensor could then preprocess this data, package it, and forward the vector to the neurons and neural structures it's connected to.

After the NN has processed the sensory data, the actuator will have received the output signals from all the neurons connecting to it. The actuator could then postprocess the accumulated vector and, if it is the type of actuator that interfaces with a scape, it will then send this scape the {Actuator_PId, actuator_name, Action} message. At this point the actuator could be finished, and then inform the cortex of it by sending it the {self(), sync} message. But we could also, instead of making our actuator immediately send the cortex a sync message, take this opportunity to make the actuator wait for the scape to send it some message. For example at this point, based on the actuator’s action, the scape could send back to it a fitness score message. Or it could send the NN a message telling it that the simulated robot had just exploded. If the scape uses this opportunity to send some fitness based information back to the actuator, the actuator could then, instead of simply sending the {self(), sync} message to the cortex, send a message containing the fitness points it was rewarded with by the scape, and whether the scape has notified it that the simulation or some training session has just ended. For example, in the case where the simulated organism dies within the simulated environment, or in the case where the NN has solved or won the game... there needs to be a way for the scape to notify the NN that something significant had just happened. This approach could be a way to do just that.

What makes the actuator a perfect process to which the scape should send this information, is: 1. Because each separate actuator belonging to the same NN could be interfacing with a different scape and so different scapes could each send to the cortex a message through its own interfacing actuator, and 2. Because the cortex synchronizes the sensors based on the signals it receives from actuators. This is important because if at any point one of the actuators sends it a halt message, the cortex has the chance to stop or pause the whole thing by simply not triggering the sensors to action. Thus if any of the actuators propagates the “end of the simulation/training message” from the scape to the cortex, the cortex will then know that the simulation is over, that it should not trigger the sensors to action, and that it should await new instructions from the exoself. If we were to allow the scape to contact the cortex directly, for example by sending it a message containing the gaged performance of the NN, and send it information of whether the training session is over or not, then there would be a chance that all the actuators had already contacted it, and that the cortex has already triggered its sensors to action. Using the actuators for this purpose ensures that we can stop the NN at the end of its training session and sense-think-act cycle.

Once the cortex receives some kind of halting message from the actuators, it could inform the exoself that it’s done, and pass to it the accumulated fitness points as the NN’s final fitness score. The exoself could then decide what to do next, whether to perturb the NN’s weights, or revert the weights back to their previous state... the cortex will sit and wait until the exoself decides on its next action.

We will implement the above described interface between the sensor/actuator/cortex and scape programs. Figure-7.6 shows the signal exchange steps. After exoself spawns the NN, the cortex immediately calls the sensors to action by sending them the: {CxPId,sync} messages (1). When a sensor receives the sync message from the cortex, it contacts the scape (if any) that it interfaces with, by sending it the: {SPId,ControlMsg} message (2). When the scape receives a message from a sensor, it executes the function associated with the ControlMsg, and then returns the sensory signal back to the calling sensor process. The sensory signal is sent to the sensor through the message: {ScPId,percept,SensoryVector} (3). After the sensor preprocesses (if at all) the sensory signal, it fans out the sensory vector to the neurons it is connected to (4). Then the actual neural net processes the sensory signal, (5), this is the thinking part of the NN system's sense-think-act loop. Once the neural net has processed the signal, the actuator gathers the processed signals from the output layer neurons (6). At this point the actuator does some postprocessing (if at all) of the output vector, and then executes the actuator function. The actuator function, like the sensory function, could be an action in itself, or an action message sent to some scape. In the case where the actuator is interfacing with a scape, it sends the scape a message of the form: {APid,ControlMsg,Output} (7). The scape executes the particular function associated with the ControlMsg atom, with the Output as the parameter. The executed function *IS* the action that the NN system takes within the virtual environment of the scape. After the scape executes the function that was requested by the actuator, it sends that same actuator a message: {SCPId,Fitness,HaltFlag}. This message contains the NN's complete or partial gage of fitness, and a notification if the simulation has ended, or if the avatar that the NN system controls within the scape has perished... or anything else one might wish to use the HaltFlag for (8). Finally, the actuator passes the message to the cortex in the form of a message: {APId,sync,Fitness,HaltFlag}, (9).

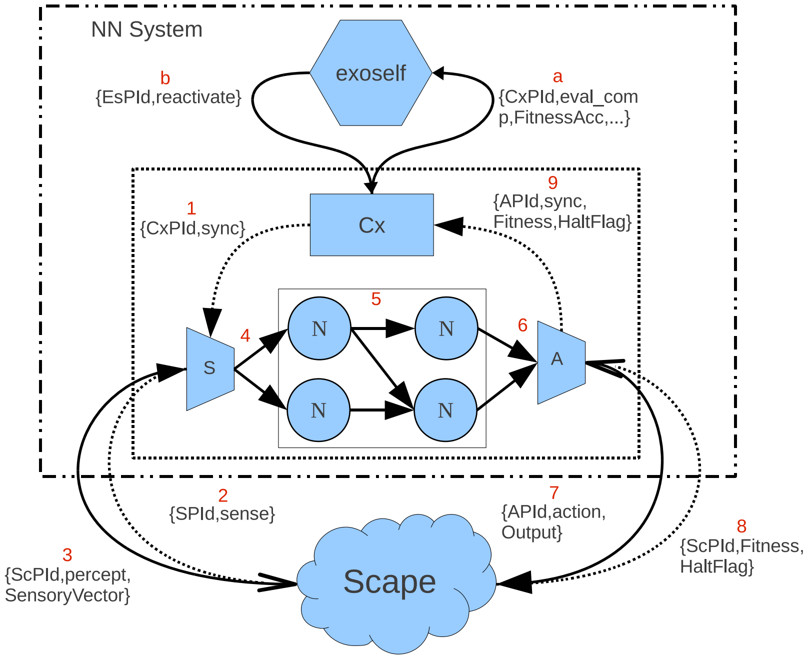
NN System
exoself
b
{EsPId,reactivate}
a
{CxPId,eval_com p,FitnessAcc,...}
Cx
1
{CxPId,sync}
9
{APId,sync, Fitness,HaltFlag}
5
S
4
N
N
N
N
6
A
2
{SPId,sense}
7
{APId,action, Output}
8
{ScPId,Fitness, HaltFlag}
3
{ScPId,percept, SensoryVector}
Scape

Fig. 7.6 The signal flow between the scape, sensors, actuators, cortex, and the exoself.

The cortex accumulates all the messages from the actuators, adding together the fitness scores, and seeing if any of the scapes have triggered the HaltFlag, which is set to 1 when triggered. If none of the HaltFlags were set to 1, then the cortex process syncs up the sensors, calling them to action, and the steps 1-9 repeat again. If any of the scapes set the HaltFlag to 1, then the cortex process takes a different course of action. At this point we could make the cortex decide whether to restart the scape, whether to start off some particular set of functions, or do any other useful task. In the version of the NN we are building in this chapter though, the XOR simulation scape will activate the HaltFlag when the training has ended, when the simulation has ran to completion. At this point the cortex simply pauses and informs the exoself process that it has finished and that the NN has been evaluated, by sending exoself the message: {CxPId, evaluation_complete, FitnessAcc, CycleAcc, TimeAcc} (a). Once again, the actions that the exoself takes are based on operational mode we chose for the system, and in the version of the system we’re building in this chapter, it will apply the SHC optimization algorithm to the NN, and then reactivate the NN system by sending cortex the following message: {ExoselfPId, reactivate} (b). After receiving the reactivation message from the exoself, the cortex’s process again loops through the steps 1-9. Note that these 9 steps represent the standard *Sense-Think-Act* cycle, where *Sense* is steps 1-4, *Think* is step 5, and *Act* is steps 6-9.

In the next section we extend the previous chapter’s architecture by developing and adding the new features we’ve discussed here.

7.6 Developing the Extended Architecture

Having discussed what new features are needed to extend the architecture so that it can be applied to various problems and simulations presented through scapes, and having decided what algorithms our system should use to optimize its neural weights, we are ready to extend the last chapter’s system. After we finish modifying the source code, our NN systems should be optimizable by the exoself & trainer processes, and be able to interact with both, public and private scapes in various manners as shown in Fig-7.7.

In the following subsections we will add the new trainer and scape modules, and modify the exoself, cortex, morphology, sensor, actuator, and neuron modules. Finally, we will also switch from *lists* to *ets* for the purpose of storing genotypes. And because we isolated the genotype reading, writing, loading, and saving functions in genotype.erl, the switch will be composed of simply modifying 4 functions to use ets instead of lists. Because we decoupled the storage and interface methods in the last chapter, it is easy for us to move to whatever storage method we find most effective for our system. Thus, modifying the genotype module to use ets instead of lists will be our first step.

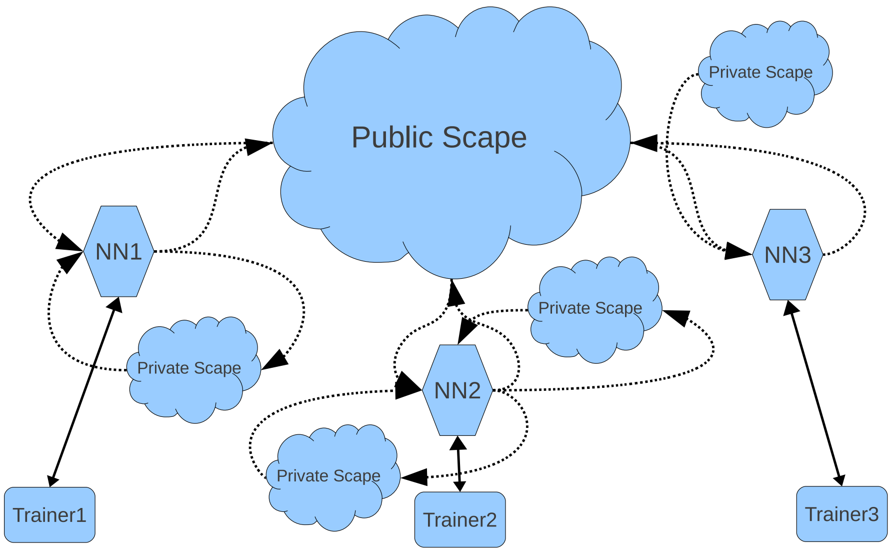

Fig. 7.7 Possible combinations of NN systems and their public/private scape interaction.

7.6.1 Modifying the genotype Module

Ets tables provide us with an efficient way to store and retrieve terms. Particularly if we need to access random terms within the table, it is better done using ets. The quick modification applied to the genotype.erl to switch from lists to ets is shown in the following listing.

Listing-7.1 Changing the genotype storage method from the simple list and file to the ets table.

```
save_genotype(FileName,Genotype)->
    TId = ets:new(FileName, [public,set,{keypos,2}]),
    [ets:insert(TId,Element) || Element <- Genotype],
    ets:tab2file(TId,FileName).
%The save_genotype/2 function expects that the Genotype is a list composed of the neuron,
sensor, actuator, cortex, and exoself elements. The function creates a new ets table, writes all
the element representing tuples from the Genotype list to the ets table, and then writes the ets
table to file.

save_to_file(Genotype,FileName)->
    ets:tab2file(Genotype,FileName).
%The save_to_file/2 function saves the ets table by the name Genotype to the file by the name
FileName.
```

```
load_from_file(FileName)->
    {ok,TId} = ets:file2tab(FileName),
    TId.
%The load_from_file/1 loads an ets representing file by the name FileName, returning the ets
table id to the caller.

read(TId,Key)->
    [R] = ets:lookup(TId,Key),
    R.
%The read/2 function reads a record associated with Key from the ets table with the id TId, re-
turning the record R to the caller. It expects that only a single record exists with the specified
Key.

write(TId,R)->
    ets:insert(TId,R).
%The function write/2 writes the record R to the ets table with the id TId.

print(FileName)->
    Genotype = load_from_file(FileName),
    Cx = read(Genotype,cortex),
    SIds = Cx#cortex.sensor_ids,
    NIds = Cx#cortex.nids,
    AIds = Cx#cortex.actuator_ids,
    io:format("~p~n",[Cx]),
    [io:format("~p~n",[read(Genotype,Id)]) || Id <- SIds],
    [io:format("~p~n",[read(Genotype,Id)]) || Id <- NIds],
    [io:format("~p~n",[read(Genotype,Id)]) || Id <- Aids].
%The function print/1 reads a stored Genotype from the file FileName, and then prints to con-
sole all the elements making up the NN's genotype.
```

The function print/1 is a new addition. It is a simple and easy way to dump the tuple encoded genotype to terminal. It is useful when you wish to see what the topology looks like, and when you need to analyze the genotype when debugging the system.

7.6.2 Modifying the morphology Module

As we discussed earlier, we want to specify the scape and its type in the sensors and actuators belonging to some particular morphology. We thus modify the records.hrl file and extend the sensor and actuator records to also contain the scape element. The new sensor and actuator records will be represented as:

```
-record(sensor,{id,name,cx_id,scape,vl,fanout_ids}).
-record(actuator,{id,name,cx_id,scape,vl,fanin_ids}).
```

In the previous chapter, we only had one type of morphology called: *test*, we now modify the morphology module by removing "test", and adding *xor_mimic*. Both, the sensor and the actuator of the xor_mimic morphology, interact with a private scape called *xor_sim*. Because the same private scape is specified for both the sensor and the actuator, only one private scape by the name xor_sim will be spawned, and they (the sensor and the actuator) will both connect to it, rather than each separately spawning its own separate xor_sim scape. The following listing shows the new xor_mimic morphology with its lists of sensors and actuators.

Listing-7.2: The new morphological type and specification added to morphology.erl

```
xor_mimic(sensors)->
   [
         #sensor{id={sensor,generate_id()}, name=xor_GetInput, scape={private,xor_sim},
vl=2}
   ];
xor_mimic(actuators)->
   [
         #actuator{id={actuator,generate_id()}, name=xor_SendOutput,
scape={private,xor_sim}, vl=1}
   ].
```

Having now modified the genotype to use ets tables, and having extended the records of our sensors and actuators, we are ready to start adding new features to our system.

7.6.3 Developing the trainer Module

When training a NN we should be able to specify all the stopping conditions, the NN based agent's morphology, and the neural network's topology. Thus the trainer process should perform the following steps:

```
1. Repeat:
   2. Create a NN of specified topology with random weights.
   3. Wait for the trained NN's final fitness
   4. Compare the trained NN's fitness to an already stored NN's fitness, if any.
   5. If the new NN is better, overwrite the old one with it.
6. Until: One of the stopping conditions is reached
7. Return: Best genotype and its fitness score.
```

We will implement all 3 types of stopping conditions:

1. A fitness score that we wish to reach,
2. The Max_Attempts that we’re willing to perform before giving up.
3. The maximum number of evaluations we’re willing to perform before giving up.

If any one of these conditions is triggered, the training ends. For default, we set maximum number of evaluations (EVAL_LIMIT) and the minimum required fitness (FITNESS_TARGET) to inf, which means that these conditions will never be reached since in Erlang an atom is considered greater than a number. Thus when starting the trainer by executing go/2, the process will default to simply using MAX_ATTEMPTS, which is set to 5. The complete source code for trainer.erl is shown in the following listing.

Listing-7.3 The implementation of the trainer module.

```
-module(trainer).
-compile(export_all).
-include(“records.hrl”).
-define(MAX_ATTEMPTS,5).
-define(EVAL_LIMIT,inf).
-define(FITNESS_TARGET,inf).

go(Morphology,HiddenLayerDensities)->
    go(Morphology,HiddenLayerDensities,?MAX_ATTEMPTS,?EVAL_LIMIT,
?FITNESS_TARGET).
go(Morphology,HiddenLayerDensities,MaxAttempts,EvalLimit,FitnessTarget)->
    PId = spawn(trainer,loop,[Morphology,HiddenLayerDensities,FitnessTarget,
{1,MaxAttempts},{0,EvalLimit},{0,best},experimental]),
    register(trainer,PId).
%The function go/2 is executed to start the training process based on the Morphology and
HiddenLayerDensities specified. The go/2 function uses a default values for the Max_Attempts,
Eval_Limit, and Fitness_Target parameters, which makes the training based purely on the
Max_Attempts value. Function go/5 allows for all the stopping conditions to be specified.

loop(Morphology,_HLD,FT,{AttemptAcc,MA},{EvalAcc,EL},{BestFitness,BestG},_ExpG,
CAcc,TAcc) when (AttemptAcc>=MA) or (EvalAcc>=EL) or (BestFitness>=FT)->
    genotype:print(BestG),
    io:format(“ Morphology:~p Best Fitness:~p EvalAcc:~p~n”, [Morphology, BestFitness,
EvalAcc]);
loop(Morphology,HLD,FT,{AttemptAcc,MA},{EvalAcc,EvalLimit},{BestFitness,BestG},
ExpG,CAcc,TAcc)->
    genotype:construct(ExpG,Morphology,HLD),
    Agent_PId=exoself:map(ExpG),
    receive
            {Agent_PId,Fitness,Evals,Cycles,Time}->
                    U_EvalAcc = EvalAcc+Evals,
                    U_CAcc = CAcc+Cycles,
                    U_TAcc = TAcc+Time,
                    case Fitness > BestFitness of
                            true ->
                                    file:rename(ExpG,BestG),
                                    ?MODULE:loop(Morphology,HLD,FT,{1,MA}, {U_EvalAcc,
EvalLimit}, {Fitness,BestG},ExpG,U_CAcc,U_TAcc);
                            false ->
                                    ?MODULE:loop(Morphology,HLD,FT,{AttemptAcc+1,MA},
{U_EvalAcc,EvalLimit}, {BestFitness,BestG}, ExpG,U_CAcc,U_TAcc)
                    end;
            terminate ->
                    io:format("Trainer Terminated:~n"),
                    genotype:print(BestG),
                    io:format(" Morphology:~p Best Fitness:~p EvalAcc:~p~n", [Morphology,
BestFitness,EvalAcc])
    end.
%loop/7 generates new NNs and trains them until a stopping condition is reached. Once any
one of the stopping conditions is reached, the trainer prints to screen the genotype, the morpho-
logical name of the organism being trained, the best fitness score achieved, and the number of
evaluations taken to find this fitness score.
```

7.6.4 Modifying the exoself Module

The Exoself's purpose is to train and monitor the NN system. Though at this point we will only implement the NN activation, training, and termination. But the way we will design the exoself process, and its general position in the NN based agent's architecture, will allow it to be modified (and we eventually will modify it) to support the NN's fault tolerance and self repair functionality. In the following listing we add to the exoself module the necessary source code it needs to train the NN system using the augmented SHC algorithm we covered in section 7.1.

```
Listing-7.4: Modifications added to the exoself.erl

prep(FileName,Genotype)->
    {V1,V2,V3} = now(),
    random:seed(V1,V2,V3),
    IdsNPIds = ets:new(idsNpids,[set,private]),
    Cx = genotype:read(Genotype,cortex),
    Sensor_Ids = Cx#cortex.sensor_ids,
    Actuator_Ids = Cx#cortex.actuator_ids,
    NIds = Cx#cortex.nids,
    ScapePIds=spawn_Scapes(IdsNPIds,Genotype,Sensor_Ids,Actuator_Ids),
    spawn_CerebralUnits(IdsNPIds,cortex,[Cx#cortex.id]),
    spawn_CerebralUnits(IdsNPIds,sensor,Sensor_Ids),
    spawn_CerebralUnits(IdsNPIds,actuator,Actuator_Ids),
    spawn_CerebralUnits(IdsNPIds,neuron,NIds),
    link_Sensors(Genotype,Sensor_Ids,IdsNPIds),
    link_Actuators(Genotype,Actuator_Ids,IdsNPIds),
    link_Neurons(Genotype,NIds,IdsNPIds),
    {SPIds,NPIds,APIds}=link_Cortex(Cx,IdsNPIds),
    Cx_PId = ets:lookup_element(IdsNPIds,Cx#cortex.id,2),
    loop(FileName,Genotype,IdsNPIds,Cx_PId,SPIds,NPIds,APIds,ScapePIds,0,0,0,0,1).
%Once the FileName and the Genotype are dropped into the prep/2 function, the function uses
the current time to create a new random seed. Then the cortex is extracted from the genotype
and the Sensor, Actuator, and Neural Ids are extracted from it. The sensors and actuators are
dropped into the spawn_Scapes/4, which extracts the scapes that need to be spawned, and then
spawns them. Afterwards, the sensor, actuator, neuron, and the cortex elements are spawned.
Then the exoself process sends these spawned elements the PIds of the elements they are con-
nected to, thus linking all the elements together into a proper interconnected structure. The cor-
tex element is the last one to be linked, because once it receives the message from the exoself
with all the data, it immediately starts synchronizing the NN by prompting the sensors to action.
Afterwards, prep/2 drops into the exoself’s main process loop.

loop(FileName,Genotype,IdsNPIds,Cx_PId,SPIds,NPIds,APIds,ScapePIds,HighestFitness,
EvalAcc,CycleAcc,TimeAcc,Attempt)->
    receive
            {Cx_PId,evaluation_completed,Fitness,Cycles,Time}->
                    {U_HighestFitness,U_Attempt}=case Fitness > HighestFitness of
                            true ->
                                    [NPId ! {self(),weight_backup} || NPId <- NPIds],
                                    {Fitness,0};
                            false ->
                                    Perturbed_NPIds=get(perturbed),
                                    [NPId ! {self(),weight_restore} || NPId <- Perturbed_NPIds],
                                    {HighestFitness,Attempt+1}
                    end,
                    case U_Attempt >= ?MAX_ATTEMPTS of
                            true ->%End training
                                    U_CycleAcc = CycleAcc+Cycles,
                                    U_TimeAcc = TimeAcc+Time,
                                    backup_genotype(FileName,IdsNPIds,Genotype,NPIds),
                                    terminate_phenotype(Cx_PId,SPIds,NPIds,APIds,ScapePIds),
                                    io:format(“Cortex:~p finished training. Genotype has been
backed up.~n Fitness:~p~n TotEvaluations:~p~n TotCycles:~p~n TimeAcc:~p~n", [Cx_PId,
U_HighestFitness, EvalAcc, U_CycleAcc, U_TimeAcc]),
                                    case whereis(trainer) of
                                         undefined ->
                                              ok;
                                         PId ->
                                              PId!{self(),U_HighestFitness, EvalAcc,
U_CycleAcc, U_TimeAcc}
                                    end;
                            false -> %Continue training
                                    Tot_Neurons = length(NPIds),
                                    MP = 1/math:sqrt(Tot_Neurons),
                                    Perturb_NPIds=[NPId || NPId <- NPIds,random:uniform()<MP],
                                    put(perturbed,Perturb_NPIds),
                                    [NPId ! {self(),weight_perturb} || NPId <- Perturb_NPIds],
                                    Cx_PId ! {self(),reactivate},
                                    loop(FileName,Genotype, IdsNPIds,Cx_PId,SPIds, NPIds,APIds,
ScapePIds,U_HighestFitness, EvalAcc+1, CycleAcc+Cycles, TimeAcc+Time, U_Attempt)
                    end
    end.
%The main process loop waits for the NN to complete the task, receive its fitness score, and
send Exoself the: {Cx_PId,evaluation_completed,Fitness,Cycles,Time} message. The message
contains all the information about that particular evaluation, the acquired fitness score, the
number of total Sense-Think-Act cycles executed, and the time it took to complete the evalua-
tion. The exoself then compares the Fitness to the one it has on record (if any), and based on
that decides whether to revert the previously perturbed neurons back to their original state or
not. If the new Fitness is lower, then the perturbed neurons are contacted and their weights are
reverted. If the new Fitness is greater than the one stored on record, then the NN is backed up to
file, and the variable EvalAcc is reset to 0. Finally, depending on whether the NN has failed to
improve its fitness Max_Attempts number of times, the exoself decides whether another NN
perturbation attempt is warranted. If it is warranted, then the exoself chooses which neurons to
mutate by randomly choosing each neuron with the probability of 1/sqrt(Tot_Neurons), where
Tot_Neurons is the total number of neurons in the neural network. The exoself saves the PIds of
those chosen neurons to process dictionary, and then sends those neurons a signal that they
should perturb their weights. Finally it tells cortex to reactivate and start syncing the sensors
and actuators again. But if the NN has failed to improve its fitness for Max_Attempts number
of times, if EvalAcc > Max_Attempts, then the exoself terminates all the elements in the NN,
and if there is a registered process by the name 'trainer', the exoself sends it the HighestFitness
score that its NN achieved and the number of total evaluations it took to achieve it.

    spawn_Scapes(IdsNPIds,Genotype,Sensor_Ids,Actuator_Ids)->
            Sensor_Scapes = [(genotype:read(Genotype,Id))#sensor.scape || Id<-Sensor_Ids],
            Actuator_Scapes = [(genotype:read(Genotype,Id))#actuator.scape || Id<-
```

```
Actuator_Ids],
          Unique_Scapes = Sensor_Scapes++(Actuator_Scapes--Sensor_Scapes),
          SN_Tuples=[{scape:gen(self(),node()),ScapeName} || {private,ScapeName}<-
Unique_Scapes],
          [ets:insert(IdsNPIds,{ScapeName,PId}) || {PId,ScapeName} <- SN_Tuples],
          [ets:insert(IdsNPIds,{PId,ScapeName}) || {PId,ScapeName} <-SN_Tuples],
          [PId ! {self(),ScapeName} || {PId,ScapeName} <- SN_Tuples],
          [PId || {PId,_ScapeName} <-SN_Tuples].
%spawn_Scapes/4 first extracts all the scape names from sensors and actuators, then builds a
list of unique scapes, and then finally extracts and spawns the private scapes. The public scapes
are not spawned since they are independent of the NN, and should already be running. The rea-
son for extracting the list of unique scapes is because if both, a sensor and an actuator are point-
ing to the same scape, then that means that they will interface with the same scape, and it does
not mean that each one should spawn its own scape of the same name. Afterwards we use the
IdsNPids ETS table to create a map from scape PId to scape name, and from scape name to
scape PId for later use. The function then sends each spawned scape a message composed of
the exoself's PId, and the scape's name: {self(),ScapeName}. Finally, a spawned scape PId list
is composed and returned to the caller.
```

We also modify the *terminate_Phenotype* function to also accept the ScapePIds parameter, and terminate all the scapes before terminating the Cortex process. Thus the following line is added to the function:

```
[PId ! {self(),terminate} || PId <- ScapePIds]
```

Having created the function which extracts the names of the scapes from the sensors and actuators of the NN system, we now develop our first scape and the very first problem on which we'll test our learning algorithm on.

7.6.5 Developing the scape Module

Looking ahead, we will certainly apply our Neuroevolutionary system to many different problems. Our system should be able to deal with ALife problems as easily as with pole balancing, financial trading, circuit design & optimization, image analysis, or any other of the infinite problems that exist. It is for this reason that we've made the sensors/actuators/scape a separate part from the NN itself, all specified through the morphology module. This way, for every problem we wish to apply our neuroevolutionary system to, we can keep the NN specific modules the same, and simply create a new morphology/scape packages.

Because there will be many scapes, a new scape for almost every problem that we'll want to solve or apply our neuroevolutionary system to, we will use the same scape.erl module and specify the different scapes within it by function name. Because the first and standard problem to apply a NN to is the XOR (Exclusive-OR) problem, our first scape will be a scape called xor_sim. The xor problem is the "hello world" of NN problems. The goal is to teach a NN to act like a XOR operation. The truth table of the 2 input XOR operation is presented in the following listing.

Listing-7.5: Truth table of the XOR operation.

X1	X2	X1 XOR X2
false	true	true
false	false	false
true	false	true
true	true	false

Because the range of tanh, the activation function of our neuron, is between -1 and 1, we will represent false as -1, and true as 1 (a bipolar encoding), rather than 0 and 1 (a unipolar encoding). This way we can use the full output range of our neurons. The following listing shows the complete source code of the scape module, which at this point contains only the single scape named xor_sim.

Listing-7.6: The complete scape module.

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

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

prep(ExoSelf_PId) ->
    receive
            {ExoSelf_PId,Name} ->
                    scape:Name(ExoSelf_PId)
    end.
%gen/2 is executed by the exoself. The function spawns prep/1 process, and awaits the name of
the scape from the exoself. Each scape is a separate and independent process, a self contained
system that was developed to interface with the sensors and actuators from which its name was
extracted. The name of the scape is the name of its main process loop.
```

```
xor_sim(ExoSelf_PId)->
   XOR = [{[-1,-1],[-1]},{[1,-1],[1]},{[-1,1],[1]},{[1,1],[-1]}],
   xor_sim(ExoSelf_PId,{XOR,XOR},0).

xor_sim(ExoSelf_PId,{[{Input,CorrectOutput}|XOR],MXOR},ErrAcc) ->
   receive
         {From,sense} ->
               From ! {self(),percept,Input},
               xor_sim(ExoSelf_PId,{[{Input,CorrectOutput}|XOR],MXOR},ErrAcc);
         {From,action,Output}->
               Error = list_compare(Output,CorrectOutput,0),
               case XOR of
                     [] ->
                           MSE = math:sqrt(ErrAcc+Error),
                           Fitness = 1/(MSE+0.00001),
                           From ! {self(),Fitness,1},
                           xor_sim(ExoSelf_PId,{MXOR,MXOR},0);
                     _ ->
                           From ! {self(),0,0},
                           xor_sim(ExoSelf_PId,{XOR,MXOR},ErrAcc+Error)
               end;
         {ExoSelf_PId,terminate}->
               ok
   end.

   list_compare([X|List1],[Y|List2],ErrorAcc)->
         list_compare(List1,List2,ErrorAcc+math:pow(X-Y,2));
   list_compare([],[],ErrorAcc)->
         math:sqrt(ErrorAcc).
```

%xor_sim/3 is a scape that simulates the XOR operation, interacts with the NN, and gages the NN’s performance. xor_sim expects two types of messages from the NN, one message from the sensor and one from the actuator. The message: {From,sense} prompts the scape to send the NN the percept, which is a vector of length 2 and contains the XOR input. The second expected message from the NN is the message from the actuator, which is expected to be an output of the NN and packaged into the form: {From,action,Output}. At this point xor_sim/3 compares the Output with the expected output that is associated with the sensory message that should have been gathered by the sensors, and then sends back to the actuator process a message composed of the scape’s PId, Fitness, and a HaltFlag which specifies whether the simulation has ended for the NN. The scape keeps track of the Mean Squared Error between the NN’s output and the correct output. Once the NN has processed all 4 signals for the XOR, the scape computes the total MSE, converts it to fitness, and finally forwards this fitness and the HaltFlag=1 to the NN. This particular scape uses the lifetime based fitness, rather than step-based fitness. During all the other steps the scape sends the actuator the signal: {Scape_PId,0,0}, while it accumulates the errors, and only at the very end does it calculate the total fitness, which is the inverse of the error with a small extra added value to avoid the divide by 0 errors. Afterwards, xor_sim resets back to its initial state and awaits anew for signals from the NN.

7.6.6 Modifying the cortex Module

As in the previous chapter, the cortex element still acts as the synchronizer of the sensors and actuators. But now, it will also keep track of the accumulated fitness score, propagated to it by the actuators. The cortex will also keep track of whether the *HaltFlag==1* was sent to it by any of the actuators, which would signify that the cortex element should halt, notify its exoself of the achieved fitness score, and then await for further instructions from it. Finally, the cortex will also keep track of the number of sense-think-act cycles performed during its lifetime. The augmented cortex module is shown in the following listing.

Listing-7.7 The updated cortex module.

```
-module(cortex).
-compile(export_all).
-include("records.hrl").
-record(state,{id,exoself_pid,spids,npids,apids,cycle_acc=0,fitness_acc=0,endflag=0,status}).

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

prep(ExoSelf_PId) ->
    {V1,V2,V3} = now(),
    random:seed(V1,V2,V3),
    receive
            {ExoSelf_PId,Id,SPIds,NPIds,APIds} ->
                    put(start_time,now()),
                    [SPId ! {self(),sync} || SPId <- SPIds],
                    loop(Id,ExoSelf_PId,SPIds,{APIds,APIds},NPIds,1,0,0,active)
    end.
%The gen/2 function spawns the cortex element, which immediately starts to wait for its initial
state message from the same process that spawned it, exoself. The initial state message contains
the sensor, actuator, and neuron PId lists. Before dropping into the main loop, CycleAcc,
FitnessAcc, and HFAcc (HaltFlag Acc), are all set to 0, and the status of the cortex is set to ac-
tive, prompting it to begin the synchronization process and call the sensors to action.

loop(Id, ExoSelf_PId, SPIds, {[APId|APIds], MAPIds}, NPIds, CycleAcc, FitnessAcc, HFAcc,
active) ->
    receive
            {APId,sync,Fitness,HaltFlag} ->
                    loop(Id,ExoSelf_PId,SPIds,{APIds,MAPIds},NPIds,CycleAcc,FitnessAcc+
Fitness, HFAcc+HaltFlag,active);
            terminate ->
                    io:format("Cortex:~p is terminating.~n",[Id]),
                    [PId ! {self(),terminate} || PId <- SPIds],
                    [PId ! {self(),terminate} || PId <- MAPIds],
                    [PId ! {self(),termiante} || PId <- NPIds]
    end;
loop(Id,ExoSelf_PId,SPIds,{[],MAPIds},NPIds,CycleAcc,FitnessAcc,HFAcc,active)->
    case EFAcc > 0 of
            true ->%Organism finished evaluation
                    TimeDif=timer:now_diff(now(),get(start_time)),
                    ExoSelf_PId ! {self(),evaluation_completed,FitnessAcc,CycleAcc,TimeDif},
                    loop(Id,ExoSelf_PId,SPIds,{MAPIds,MAPIds},NPIds,CycleAcc,FitnessAcc,
HFAcc, inactive);
            false ->
                    [PId ! {self(),sync} || PId <- SPIds],

    loop(Id,ExoSelf_PId,SPIds,{MAPIds,MAPIds},NPIds,CycleAcc+1,FitnessAcc,
HFAcc,active)
    end;
loop(Id, ExoSelf_PId, SPIds, {MAPIds,MAPIds}, NPIds, _CycleAcc, _FitnessAcc, _HFAcc,
inactive)->
    receive
            {ExoSelf_PId,reactivate}->
                    put(start_time,now()),
                    [SPId ! {self(),sync} || SPId <- SPIds],
                    loop(Id,ExoSelf_PId,SPIds,{MAPIds,MAPIds},NPIds,1,0,0,active);
            {ExoSelf_PId,terminate}->
                    ok
    end.
%The cortex's goal is to synchronize the NN system's sensors and actuators. When the actua-
tors have received all their control signals, they forward the sync messages, the Fitness, and the
HaltFlag messages to the cortex. The cortex accumulates these Fitness and HaltFlag signals,
and if any of the HaltFlag signals have been set to 1, HFAcc will be greater than 0, signifying
that the cortex should halt. When EFAcc > 0, the cortex calculates the total amount of time it
has ran (TimeDiff), and forwards to exoself the values: FitnessAcc, CycleAcc, and TimeDiff.
Afterwards, the cortex enters the inactive mode and awaits further instructions from the exoself.
If none of the HaltFlags were set to 0, then the value HFAcc == 0, and the cortex triggers off
another Sense-Think-Act cycle. The reason the cortex process stores 2 copies of the actuator
PIds: the APIds, and the MemoryAPIds (MAPIds), is so that once all the actuators have sent it
the sync messages, it can restore the APIds list from the MAPIds.
```

7.6.7 Modifying the neuron Module

To allow the exoself process to optimize the weights of the NN through the SHC algorithm, we will need to give our neurons the ability to have their weights perturbed when requested to do so by the exoself, and have their weights reverted when/if requested by the same. We will also specify the range of these weight perturbations in this module, and the weight saturation values, the maximum and minimum values that the weights can take. It is usually not a good idea to let the weights reach very large positive or negative values, as that would allow any single weight to completely overwhelm other synaptic weights of the same neuron. For example if a neuron has 100 weights in total, and one of the weights has a value of 1000000, no other weight can compete with it unless it too is raised to such a high value. This results in a single weight controlling the information processing ability of the entire neuron. It is important that no weight can overwhelm all others (which prevents the neuron from performing coherent processing of signals), for this reason we will set the saturation limit to 2*Pi, and the perturbation intensity to half that. The perturbation intensity is half the value of weight saturation point so that there will always be a chance that the weight could flip from a positive to a negative value. The range of the weight perturbation intensity is specified by the DELTA_MULTIPLIER macro as: -define(DELTA_MULTIPLIER,math:pi()*2), since we will multiply it by (random:uniform()-0.5), the actual range will be between -Pi and Pi.

The complete neuron module is shown in the following listing.

Listing-7.8 The neuron module.

```
-module(neuron).
-compile(export_all).
-include("records.hrl").
-define(DELTA_MULTIPLIER,math:pi()*2).
-define(SAT_LIMIT,math:pi()*2).

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

prep(ExoSelf_PId) ->
   {V1,V2,V3} = now(),
   random:seed(V1,V2,V3),
   receive
         {ExoSelf_PId,{Id,Cx_PId,AF,Input_PIdPs,Output_PIds}} ->
               loop(Id,ExoSelf_PId,Cx_PId,AF,{Input_PIdPs,Input_PIdPs},Output_PIds,0)
   end.
%When gen/2 is executed it spawns the neuron element, which seeds the pseudo random number generator, and immediately begins to wait for its initial state message. It is essential that we seed the random number generator to make sure that every NN will have a different set of mutation probabilities and different combination of perturbation intensities. Once the initial state signal from the exoself is received, the neuron drops into its main loop.
```

```
loop(Id,ExoSelf_PId,Cx_PId,AF,{[{Input_PId,Weights}|Input_PIdPs],MInput_PIdPs},Output_
PIds,Acc)->
    receive
            {Input_PId,forward,Input}->
                    Result = dot(Input,Weights,0),

    loop(Id,ExoSelf_PId,Cx_PId,AF,{Input_PIdPs,MInput_PIdPs},Output_PIds,Result+Acc);
            {ExoSelf_PId,weight_backup}->
                    put(weights,MInput_PIdPs),

    loop(Id,ExoSelf_PId,Cx_PId,AF,{[{Input_PId,Weights}|Input_PIdPs],MInput_PIdPs},Outp
ut_PIds,Acc);
            {ExoSelf_PId,weight_restore}->
                    RInput_PIdPs = get(weights),

    loop(Id,ExoSelf_PId,Cx_PId,AF,{RInput_PIdPs,RInput_PIdPs},Output_PIds,Acc);
            {ExoSelf_PId,weight_perturb}->
                    PInput_PIdPs=perturb_IPIdPs(MInput_PIdPs),

    loop(Id,ExoSelf_PId,Cx_PId,AF,{PInput_PIdPs,PInput_PIdPs},Output_PIds,Acc);
            {ExoSelf_PId,get_backup}->
                    ExoSelf_PId ! {self(),Id,MInput_PIdPs},

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

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

```
%The neuron process waits for vector signals from all the processes that it's connected from. As the presynaptic signals fanin, the neuron takes the dot product of the input and their associated weight vectors, and then adds it to the accumulator. Once all the signals from Input_PIds are received, the accumulator contains the dot product to which the neuron then adds the bias (if it exists) and executes the activation function. After fanning out the output signal, the neuron again returns to waiting for incoming signals. When the neuron receives the {ExoSelf_PId, get_backup} message, it forwards to the exoself its full MInput_PIdPs list, and its Id. The MInput_PIdPs contains the current version of the neural weights. When the neuron receives the {ExoSelf_PId,weight_perturb} message, it executes the perturb_IPIdPs/1, after which the neuron drops back into the loop but with MInput_PIdPs replaced by the new PInput_PIdPs. It is important to note that the neuron expects to be synchronized, and expects that it has at this point not received any signals from the other elements it is connected from, because if it has and it then changes out the Input_PIdPs with PInput_PIdPs, it might start waiting for signals from the elements from which it has already received the signals. When the neuron receives the {ExoSelf_PId,weight_backup}, it stores its weights in its process dictionary. When the neuron receives the {ExoSelf,weight_restore}, it restores its weights to the state they were before being perturbed by restoring the saved synaptic weights from its process dictionary.

   tanh(Val)->
         math:tanh(Val).
%The activation function is a sigmoid function, tanh.

perturb_IPIdPs(Input_PIdPs)->
   Tot_Weights=lists:sum([length(Weights) || {_Input_PId,Weights}<-Input_PIdPs]),
   MP = 1/math:sqrt(Tot_Weights),
   perturb_IPIdPs(MP,Input_PIdPs,[]).
perturb_IPIdPs(MP,[{Input_PId,Weights}|Input_PIdPs],Acc)->
   U_Weights = perturb_weights(MP,Weights,[]),
   perturb_IPIdPs(MP,Input_PIdPs,[{Input_PId,U_Weights}|Acc]);
perturb_IPIdPs(MP,[Bias],Acc)->
   U_Bias = case random:uniform() < MP of
         true-> sat((random:uniform()-0.5)*?DELTA_MULTIPLIER+Bias,-
?SAT_LIMIT,?SAT_LIMIT);
         false -> Bias
   end,
   lists:reverse([U_Bias|Acc]);
perturb_IPIdPs(_MP,[],Acc)->
   lists:reverse(Acc).
%perturb_IPIdPs/1 first calculates the probability that a weight will be perturbed, the probability being the inverse square root of the total number of weights in the neuron. The function then drops into perturb_IPIdPs/3, which executes perturb_weights/3 for every set of weights associated with a particular Input_PId in the Input_PIdPs list. If bias is present in the weights list, it is
```

reached last and perturbed just as any other weight, based on the probability. Afterwards, the perturbed and inverted version of the Input_PIdPs is reversed back to the proper order and returned to the calling function.

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

            sat(Val,Min,Max)->
                    if
                            Val < Min -> Min;
                            Val > Max -> Max;
                            true -> Val
                    end.
```

%perturb_weights/3 accepts a probability value, a list of weights, and an empty list to act as an accumulator. The function then goes through the weight list perturbing each weight with a probability of MP. The weights are constrained to be within the range of -?SAT_LIMIT and SAT_LIMIT through the use of the sat/3 function.

7.6.8 Modifying the sensor Module

We have already created the xor_sim scape which expects a particular set of messages from the sensors and actuators interfacing with it. The scape expects the message: {Sensor_PId,sense} from the sensor, to which it responds with the sensory data sent in the: {Scape_PId,percept,SensoryVector} format. We now add to the sensor module a new function which can send and receive such messages. As we decided in the morphology module, the name of the new sensor will be xor_GetInput. The new function is shown in the following listing.

Listing-7.9 The implementation of the xor_GetInput sensor.

```
xor_GetInput(VL,Scape)->
    Scape ! {self(),sense},
    receive
            {Scape,percept,SensoryVector}->
                    case length(SensoryVector)==VL of
                    true ->
                              SensoryVector;
                    false ->
                              io:format("Error in sensor:xor_sim/2, VL:~p
SensoryVector:~p~n", [VL,SensoryVector]),
                              lists:duplicate(VL,0)
          end
   end.
%xor_GetInput/2 contacts the XOR simulator and requests the sensory vector, which in this
case should be a vector of length 2. The sensor checks that the incoming sensory signal, the
percept, is indeed of length 2. If the vector length differs, then this is printed to the console and
a dummy vector of appropriate length is constructed and used. This prevents unnecessary
crashes in the case of errors, and gives the researcher a chance to fix the error and hotswap the
code.
```

7.6.9 Modifying the actuator Module

The scape expects the message: {Actuator_PId,action,Output} from the actuator, to which it responds with the: {Scape_PId,FItness,EndFlag} message. We decided in the morphology module to call the actuator that will interface with the xor_sim scape: xor_SendOutput. The following listing shows this newly added function to the actuator module.

Listing-7.10 The implementation of the xor_SendOutput actuator.

```
xor_SendOutput(Output,Scape)->
   Scape ! {self(),action,Output},
   receive
          {Scape,Fitness,HaltFlag}->
                    {Fitness,HaltFlag}
   end.
%xor_SendOutput/2 function simply forwards the Output vector to the XOR simulator, and
then waits for the resulting Fitness and HaltFlag message from the scape.
```

7.7 Compiling Modules & Simulating the XOR Operation

We have now added all the new features, functions, and elements we needed to implement the learning algorithm with our NN system. When modifying our system to give it the ability to learn through the use of the augmented stochastic hill-climbing, we also implemented the xor_sim scape and the associated morphology with its sensors and actuators. We now compile all the modules we've created and modified to see if they work, and then test if our system can indeed learn. To compile everything in one step, make sure you’re in the folder where all the modules are stored, and then execute the following command:

```
1>make:all([load]).
…
up_to_date
```

Now that everything is compiled, we can test to see if our NN can learn to simulate the XOR operation. The XOR operation cannot be performed by a single neuron. To solve this problem by a strictly feed forward neural network, the minimum required topology to perform this task is: [2,1], 2 neurons in the first layer and 1 in the output layer. We can specify the morphology, the NN topology, and the stopping condition, right from the trainer. The morphology, xor_mimic, specifies the problem we wish to apply the NN to, and the sensors and actuators that will interface with the xor_sim scape which will simulate the XOR operation and gage the NN’s ability to simulate it. For the stopping condition we will choose to use fitness. We’ll decide on the minimum fitness we wish our NN to achieve by calculating the total error in the NN’s approximation of XOR that we’re willing to accept. For example, a maximum error of 0.001 translates into a minimum fitness of 1/(0.01+ 0.00001), or: 99.9. Since we do not wish to use other stopping conditions, we’ll set them to inf. Thus each training session will run until the NN can approximate XOR with an error no greater than 0.001 (a fitness no less than 99.9).

```
1>trainer:go(xor_mimic,[2],inf,inf,99.9).
Finished updating genotype to file:experimental
Cortex:<0.104.0> finished training. Genotype has been backed up.
 Fitness:188.94639182995695
 TotEvaluations:224
 TotCycles:896
{cortex,cortex,
     [{sensor,7.617035388076853e-10}],
     [{actuator,7.617035388076819e-10}],
     [{neuron,{1,7.61703538807679e-10}},
      {neuron,{1,7.617035388076778e-10}},
      {neuron,{2,7.617035388076755e-10}}]}
{sensor,{sensor,7.617035388076853e-10},
     xor_GetInput,cortex,undefined,
     {private,xor_sim},
     2,
     [{neuron,{1,7.61703538807679e-10}},{neuron,{1,7.617035388076778e-10}}],
     undefined,[],[]}
{neuron,{neuron,{1,7.61703538807679e-10}},
     cortex,tanh,
     [{{sensor,7.617035388076853e-10},
      [-6.283185307179586,6.283185307179586]},
      {bias,-6.283185307179586}],
    [{neuron,{2,7.617035388076755e-10}}]}
{neuron,{neuron,{1,7.617035388076778e-10}},
    cortex,tanh,
    [{{sensor,7.617035388076853e-10},
      [-5.663623085487123,6.283185307179586]},
      {bias,6.283185307179586}],
    [{neuron,{2,7.617035388076755e-10}}]}
{neuron,{neuron,{2,7.617035388076755e-10}},
    cortex,tanh,
    [{{neuron,{1,7.617035388076778e-10}},[-6.283185307179586]},
      {{neuron,{1,7.61703538807679e-10}},[6.283185307179586]},
      {bias,6.283185307179586}],
    [{actuator,7.617035388076819e-10}]}
{actuator,{actuator,7.617035388076819e-10},
      xor_SendOutput,cortex,undefined,
      {private,xor_sim},
      1,
      [{neuron,{2,7.617035388076755e-10}}],
      undefined,[],[]}
 Morphology:xor_mimic Best Fitness:188.94639182995695 EvalAcc:224
```

It works!. The trainer used the specified morphology to generate genotypes with the particular set of sensors and actuators to interface with the scape, and eventually produced a trained NN. In this particular case, the best fitness was 188.9, and it took only 224 evaluations to reach it. The trainer also used the genotype:print/1 function to print the genotype topology to screen when it was done, which allows us to now analyze the genotype and double check it for accuracy.

Since the learning algorithm is stochastic, the number of evaluations it takes will differ from one attempt to another, some will go as high as a few thousand, other will stay in the hundreds. But if you've attempted this exercise, you might have also noted that you got roughly the same fitness, and this requires an explanation.

We're using tanh as the activation function. This activation function has *1* and -*1* as its limit, and because we're using the weight saturation limit set to *2*PI*, the neuron's output can only get so close to -1 and 1 through the tanh function, and hence the final error in the XOR operation approximation. Thus the fitness we can achieve is limited by *how close tanh(PI*2) can get to 1*, *and tanh(-PI*2) can get to -1*. Since you and I are using the same SAT_LIMIT parameters, we can achieve the same maximum fitness scores, which is what we saw in the above result: (188.9), when using the SAT_LIMIT = 2*PI. If we for example modify the SAT_LIMIT in our neuron module as follows:

```
From: -define(SAT_LIMIT,math:pi()*2)
To: -define(SAT_LIMIT,math:pi()*20)
```

Then recompile and apply the NN to the problem again, then the best fitness will come out to be *99999.99*. Nevertheless, the SAT_LIMIT equaling to *math:pi()*2* is high enough for most situations, and as noted, when we allow the weights to take a value of any magnitude, we run the risk of any one synaptic weight to overwhelm the whole neuron and make it essentially useless. Also, returning the neural weight that has ran afoul and exploded in magnitude back to an appropriate value would be difficult with small perturbations... Throughout the years I've found that having the SAT_LIMIT set to *math:pi()* or *math:pi()*2* is the most effective choice.

We've now created a completely functional, static feed forward neural network system that can be applied to a lot of different problem types through the use of the system's *scape*, *sensors*, and *actuators* packages. It's an excellent start, and because our learning algorithm is unsupervised, we can even use our NNs as controllers in ALife. But there is also a problem, our NN system is only feedforward and so it does not possess memory, achievable through recursive connections. Also, our system only uses the *tanh* activation function, which might make problems like fourier analysis, fourier synthesis, and many other problems difficult to tackle. Our system can achieve an even greater flexibility if it can use other activation functions, and just like randomly choosing synaptic weights during neuron creation, it should be able to randomly choose activation functions from some predetermined list of said functions. Another problem we've noticed even when solving the XOR problem is that we had to know the minimal topology beforehand. If in the previous problem we would have chosen a *topology of: [1] or [2],* our NN would not have been able to solve that problem. Thus our NN has a flaw, the flaw is that we need to know the proper topology, or the minimal topology which can solve the problem we're applying our NN to. We need to devise a plan to overcome this problem, by letting our NN evolve topology as well. Another problem is that even though our NN "learns", it actually does not. It is, in reality, just being optimized by an external process, the exoself. What is missing is neural plasticity, the ability of the neurons to self modify based on sensory signals, and past experience... that would be true learning, learning as self modification based on interaction with the environment within the lifetime of the organism. Finally, even though our system does have all the necessary features for us to start implementing supervision trees and process monitoring, we will first implement and develop the neuroevolutionary functionality, before transforming our system into a truly fault tolerant distributed CI system.

Before we continue on to the next chapter where we will start adding some of these mentioned features, and finally move to a population based approach and topological evolution, we first need to create a small benchmarking function. For example, we've used the trainer to solve the XOR problem and we noted that it took K number of evaluations to solve it. When you've solved it with a trainer on your computer, you probably have gotten another value... we need to create a function that automates the process of applying the trainer to some problem many times, and then calculates the average performance of the system. We need a benchmarking method because it will allow us to calculate dependable average performance of our system and allow us to compare it to other machine learning approaches. It will also allow us to test new features and get a good idea of what affect they have on our system's performance across the spectrum of problems we might have in our benchmark suit. Thus, in the next section we will develop a small benchmarking function.

7.8 Adding the benchmarker Module

In this section we want to create a small system that performs benchmarking of the NN system we've developed, on some problem or problem set. This benchmarking system should be able to summon the trainer process X number of times, applying it to some problem of our choosing. After the *benchmarker* process has spawned the trainer, it should wait for it to finish optimizing the NN system. At some point the trainer will reach its stopping condition, and then send the benchmarking process the various performance statistics of the optimization run. For example the trainer could send to the benchmarker the number of evaluations it took to train the NN to solve some problem, the amount of time it took it, and the NN size required to solve it. The benchmarker should accumulate a list of these values, and once it has applied the trainer to the chosen problem X number of times, it should calculate the averages and standard deviations of these various performance statistics. Finally, our benchmarking system should print these performance results to console. At this point the researcher could use this performance data to for example compare his system to other state of the art machine learning algorithms, or the researcher could vary some parameter or add some new features to his system and benchmark it again, and in this manner see if the new features make the system more or less effective. Thus our *benchmarker* should perform the following steps:

1. **Repeat:**
    2. Apply the trainer to some experiment or problem.
    3. Receive from the trainer the resulting data (total evaluations, total cycles, NN size...).
    4. Add this data to the statistics accumulator.
5. **Until**: The trainer has been applied to the given problem, X number of times.
6. **Return**: Calculate averages and standard deviations of the various features in the accumulator.

The following listing shows the implementation of the benchmarker module.

Listing-7.11: The implementation of the benchmarker module.

```
-module(benchmarker).
-compile(export_all).
-include("records.hrl").
-define(MAX_ATTEMPTS,5).
-define(EVAL_LIMIT,inf).
-define(FITNESS_TARGET,inf).
-define(TOT_RUNS,100).
-define(MORPHOLOGY,xor_mimic).

go(Morphology,HiddenLayerDensities)->
    go(Morphology,HiddenLayerDensities,?TOT_RUNS).
go(Morphology,HiddenLayerDensities,TotRuns)->
    go(Morphology,HiddenLayerDensities,?MAX_ATTEMPTS,?EVAL_LIMIT,
?FITNESS_TARGET,TotRuns).
go(Morphology,HiddenLayerDensities,MaxAttempts,EvalLimit,FitnessTarget,TotRuns)->
    PId = spawn(benchmarker,loop,[Morphology,HiddenLayerDensities,MaxAttempts,
EvalLimit,FitnessTarget,TotRuns,[],[],[],[]]),
    register(benchmarker,PId).
% The benchmarker is started through the go/2, go/3, or go/6 function. The parameters the
benchmark uses can be specified through the macros, and then used by executing go/2 or go/3
for which the researcher simply specifies the Morphology (the problem on which the NN will
be benchmarked) and the HiddenLayerDensities (NN topology). The go/2 and go/3 functions
execute go/6 function with default parameters. The benchmarker can also be started through
go/6, using which the researcher can manually specify all the parameters: morphology, NN to-
pology, Max Attempts, Max Evaluations, target fitness, and the total number of times to run the
trainer. Before dropping into the main loop, go/6 registers the benchmarker process so that the
trainer can send it the performance stats when it finishes.

loop(Morphology,_HiddenLayerDensities,_MA,_EL,_FT,0,FitnessAcc,EvalsAcc,CyclesAcc,
TimeAcc)->
    io:format("Benchmark results for:~p~n",[Morphology]),
    io:format("Fitness::~n Max:~p~n Min:~p~n Avg:~p~n Std:~p~n",
            [lists:max(FitnessAcc),lists:min(FitnessAcc),avg(FitnessAcc),std(FitnessAcc)]),
    io:format("Evals::~n Max:~p~n Min:~p~n Avg:~p~n Std:~p~n",
            [lists:max(EvalsAcc),lists:min(EvalsAcc),avg(EvalsAcc),std(EvalsAcc)]),
    io:format("Cycles::~n Max:~p~n Min:~p~n Avg:~p~n Std:~p~n",
            [lists:max(CyclesAcc),lists:min(CyclesAcc),avg(CyclesAcc),std(CyclesAcc)]),
    io:format("Time::~n Max:~p~n Min:~p~n Avg:~p~n Std:~p~n",
            [lists:max(TimeAcc),lists:min(TimeAcc),avg(TimeAcc),std(TimeAcc)]);
```

```
loop(Morphology,HiddenLayerDensities,MA,EL,FT,BenchmarkIndex,FitnessAcc,EvalsAcc,
CyclesAcc,TimeAcc)->
    Trainer_PId = trainer:go(Morphology,HiddenLayerDensities,MA,EL,FT),
    receive
            {Trainer_PId,Fitness,Evals,Cycles,Time}->
                    loop(Morphology,HiddenLayerDensities,MA,EL,FT,BenchmarkIndex-1,
[Fitness|FitnessAcc],[Evals|EvalsAcc],[Cycles|CyclesAcc],[Time|TimeAcc]);
            terminate ->
                    loop(Morphology,HiddenLayerDensities,MA,EL,FT,0,FitnessAcc,EvalsAcc,
CyclesAcc,TimeAcc)
    end.
% Once the benchmarker is started, it drops into its main loop. The main loop spawns the train-
er and waits for it to finish optimizing the NN system, after which it sends to the benchmarker
the performance based statistics. The benchmarker accumulates these performance statistics in
lists, rerunning the trainer TotRuns number of times. Once the benchmarker has ran the trainer
TotRuns number of times, indicated to be so when BenchmarkIndex reaches 0, it calculates the
Max, Min, Average, and Standard Deviation values for every statistic list it accumulated.

avg(List)->
    lists:sum(List)/length(List).
avg_std(List)->
    Avg = avg(List),
    std(List,Avg,[]).

    std([Val|List],Avg,Acc)->
            std(List,Avg,[math:pow(Avg-Val,2)|Acc]);
    std([],_Avg,Acc)->
            Variance = lists:sum(Acc)/length(Acc),
            math:sqrt(Variance).
%avg/1 and std/1 functions calculate the average and the standard deviation values of the lists
passed to them.
```

To make the whole system functional, we also have to slightly modify the trainer module so that when the stopping condition is reached, the trainer prints the genotype to console, unregisters itself, checks if a process by the name *benchmarker* exists, and if it does, sends it the performance stats of the optimization session. To make this modification, we add the following lines of code to our trainer module:

```
unregister(trainer),
case whereis(benchmarker) of
    undefined ->
            ok;
    PId ->
            PId ! {self(),BestFitness,EvalAcc,CAcc,TAcc}
end;
```

We now compile and recompile the benchmarker and the trainer modules respectively, and then test our new benchmarker system. To test it, we apply it to the XOR problem, executing it with the following parameters:

- Morphology: xor_mimic
- HiddenLayerDensities: [2]
- MaxAttempts: inf
- EvalLimit: inf
- FitnessTarget: 100
- TotRuns: 100

Based on these parameters, each trainer will generate genotypes until one of them solves the problem with a fitness of at least 100. Thus, the benchmarker will calculate the resulting performance statistics from 100 experiments. To start the benchmarker, execute the following command:

```
1>benchmarker:go(xor_mimic,[2],inf,inf,100,100).
…
Benchmark results for:xor_mimic
Fitness::
 Max:99999.99999999999
 Min:796.7693071321515
 Avg:96674.025859051
 Std:16508.11828048093
Evals::
 Max:2222
 Min:258
 Avg:807.1
 Std:415.8308670601546
...
```

It works! The benchmarker ran 100 training sessions and calculated averages, standard deviations, maxs, and mins for the accumulated Fitness, Evaluations, Cycles, and Time lists. Our system now has all the basic features of a solid machine learning platform.

7.9 Summary

In this chapter we added the augmented stochastic hill-climber optimization algorithm to our system, and extended the exoself process so that it can use it to tune its NN’s synaptic weights. We also developed a trainer, a system which further extends the SHC optimization algorithm by restarting genotypes when the exoself had tuned the NN’s synaptic weights and reached its stopping condition. This effectively allows the trainer process to use the Random Restart Stochastic

Hill Climbing optimization algorithm to train NNs. Finally, we created the *benchmarker* program, a system that can apply the trainer process to some problem, X number of times, and then average the performance statistics and print the results to console.

Our NN system now has all the features necessary to solve and be applied to various problems and simulations. The learning algorithm our system implements is the simple yet very powerful augmented version of the random-restart stochastic hill-climber. We also now have a standardized method of presenting simulations, training scenarios, and problems to our NN system, all through the decoupled scape packages and morphologies.

In the next chapter we will take this system even further, combining it with population based evolutionary computation and topological mutation, thus creating a simple topology and weight evolving artificial neural network system.