Chapter 10 DXNN: A Case Study

Abstract This chapter presents a case study of a memetic algorithm based TWEANN system that I developed in Erlang, called DXNN. Here we will discuss how DXNN functions, how it is implemented, and the various details and implementation choices I made while building it, and why. We also discuss the various features that it has, the features which we will eventually need to add to the system we're building together. Our system has a much cleaner and decoupled implementation, and which by the time we've reached the last chapter will supersede DXNN in every way.

Deus Ex Neural Network (DXNN) platform is the original topology and weight evolving artificial neural network system that I developed in Erlang. What you and I are creating here in this book is the next generation of it. We're developing a more decoupled version, a simpler to generalize and more refined version, and one with cleaner architecture and implementation. In this chapter we'll discuss the already existing system, how it differs from what we've created so far, and what features it has that we will in later chapters need to add to the system we've developed thus far. By the time this book ends, we'll have created not just a TWEANN system, but a Topology and Parameter Evolving Universal Learning Network framework, capable of evolving neural networks, circuits, be used as a parallel distributed genetic programming framework, posses some of the most advanced features currently known, and designed in such a way that new features can easily be added to it by simply incorporating new modules (hence the importance of developing a system where almost everything is decoupled from everything else).

DXNN is a memetic algorithm based TWEANN platform. As we discussed, the most advanced approach to neuroevolution and universal learning networks in general, is through a system that uses evolutionary algorithms to optimize both, the topology and the synaptic weights/node-parameters of the graph system. The weights and topology of a NN are evolved so as to increase the NN system's fitness, based on some fitness criteria/function.

In the following sections we will cover the algorithm and the various features that make up the DXNN system.

10.1 The Reason for the Memetic Approach to Synaptic Weight Optimization

As we have discussed in the first chapters, the standard genetic algorithm performs global and local search in a single phase, while the memetic algorithm sepa-

G.I. Sher, *Handbook of Neuroevolution Through Erlang*,
DOI 10.1007/978-1-4614-4463-3_10,

rates these two searches into separate stages. When it comes to neural networks, the global search is done through the exploration of NN topologies, and the local search is done through the optimization of synaptic weights.

Based on the benchmarks, and ALife performance of DXNN, the memetic approach has shown to be highly efficient and agile. The primary benefit of separating the two search phases is due to the importance of finding the right synaptic weights for a particular topology before deciding on the final fitness score of that topology. Standard TWEANNs typically operate using the standard genetic algorithm based mutation operator probabilities. In such systems, when creating an offspring the parent is chosen and then a single mutation operator is applied to it, with a probability of more than 97% that the mutation operator will be a synaptic weight perturbation operator. This type of operator simply selects some number of neurons and perturbs some random number of synaptic weights belonging to them The other mutation operators are the standard topology augmenting operators.

In standard TWEANNs, a system might generate an optimal topology for the problem, but because during that one innovation of the new topology the at-that-point existing synaptic weights make that topology ineffective, the new NN topology might be disregarded and removed. Also, in most TWEANNs, the synaptic weight perturbations are applied indiscriminately to all neurons of the NN, and thus if for example a NN is composed of 1 million neurons, and a new neuron is added, the synaptic weight mutations might be applied to any of the 1000001 neurons... making the probability of optimizing the new and the right neuron and its synaptic weights, very low.

As in the system we've built so far, the DXNN platform evolves new NN topologies during each generation, and then through the application of an augmented stochastic hill climbing optimizes the synaptic weights for those topologies. Thus, when the "tuning phase", which is what the local search phase is called in DXNN, has completed, the tuned NN has roughly the best set of synaptic weights for its particular topology, and thus the fitness that is given to the NN is a more accurate representation of its true performance fitness and potential.

Furthermore, the synaptic weight optimization through perturbation is not applied to all the neurons indiscriminately throughout the NN, but instead is concentrated on primarily the newly created neurons, or those neurons which have been recently affected by a mutation applied to the NN. Thus, the tuning phase optimizes the newly added neural elements so that they work and contribute positively to the NN they have been added to.

With this approach, the DXNN system is able to slowly grow and optimize the NN systems. Adding new features/elements and optimizing them to work with the already existing structures. This I believe gives DXNN a much greater ability to scale, for there is zero chance of being able to create vast neural networks when

after adding a single new neuron to a 1000000 neuron NN system, we try to then perturb random synaptic weights in hopes of somehow making the whole system cohesive and functional. Building the NN slowly, complexifying it, adding new features and ensuring that they work with the existing system in a positive way, allows us to concentrate and optimize those few newly added elements, no matter how large the already existing NN system is.

Thus, during the local search phase, during the *tuning phase*, we optimize the synaptic weights of the newly added and modified elements. And during the global search, during the *topological mutation phase*, we apply enough topological mutation operators when creating an offspring, such that we are able to create innovation in the newly resulting NN system, but few enough of them such that the newly added elements to the NN can still be optimized to work with the existing much larger, already proven system.

Having discussed the *why* behind the memetic algorithm approach taken by DXNN, we now cover the two approaches this system uses when creating offspring, clarified to a much greater detail in the next two sections. These two approaches are the *generational* evolution, and the *steady_state* evolution.

The most common approach to offspring creation, and timing of selection and mutation operator application, is *generational*. Generational evolution simply means that we create a population of some size X of seed agents, apply them to some problem, wait until all agents in the population have been evaluated and given a fitness score, then select the best of the population, allow them to create offspring, and then create the next *generation* composed of the best agents of the previous generation plus their offspring, or some other appropriate combination of fit parents and newly created offspring. That is the essence of the standard generational evolution.

The *steady state* evolution tries to emulate the biological world to a slightly greater degree. In this approach there is no wait for the entire population to be evaluated before a new agent is created. Instead, as soon as one agent has finished working on a problem (has been evaluated), or has perished or gathered enough resources (in the case of an ALife simulation), a new offspring is created. The new offspring is either created by some already existing agent through the execution of a *create_offspring* actuator, or is created by the neuroevolutionary system itself, after it has calculated what genotype/s to use as the base for the offspring creation process, and whether the environment can support another agent. In this manner, the population size working on a problem, or existing in a simulated environment, is kept relatively constant. There are always organisms in the environment, when some die, new ones are created. There is a constant turnover of new agents and new genotypes and phenotypes in the population.

Before we begin discussing the general algorithm of the *generational* and *steady_state* evolution, before we begin discussing the DXNN system and its various features, it would be helpful for me to first explain the architectures of the NN systems that are evolved by it. The DXNN's genotype encoding, and the TWEANN's architecture, differs slightly from what we've been developing in the past few chapters.

10.2 The DXNN Encoding and Architecture

The genotype encoding used by DXNN is almost exactly the same as the one used by the system we are building together. It is tuple encoded, with the tuples stored in the mnesia database. The list of records composing the genotype of each NN system in the DXNN platform is as follows:

```
-record(dx,{id,cx_id,n_ids,specie_id,constraint,morphology,generation,fitness,
profile,summary, evo_hist,mode, evo_strat}).
-record(cortex,{id,sensors,actuators,cf,ct,type,plasticity,pattern,cids,su_id,
link_form,dimensions,densities, generation}).
-record(neuron,{id,ivl,i,ovl,o,lt,ro,type,dwp,su_id,generation}).
```

The *dx* record plays the role that the *agent* record does in our TWEANN. The other thing that immediately stands out is that there are no sensor or actuator elements. If you look in the DXNN's records.hrl [1] though, you will see those records, but they are not independent elements, the sensors and actuators are part of the cortex element. Indeed in the original DXNN system, the cortex element is not a synchronization element, but a gatekeeper element. The cortex element talks directly to the neurons. The connection from the cortex to the neurons is accomplished through the *ct* list (connected to), and the signals it gathers from the neurons is done through the *cf* list (connected from). The cortex also has a sensor and actuator list, which contain the names of the sensor and actuator functions, and the lists of the neurons that they are connected to and from respectively, based on the ct/cf lists. This DXNN's NN based agent architecture is shown in Fig-10.1.

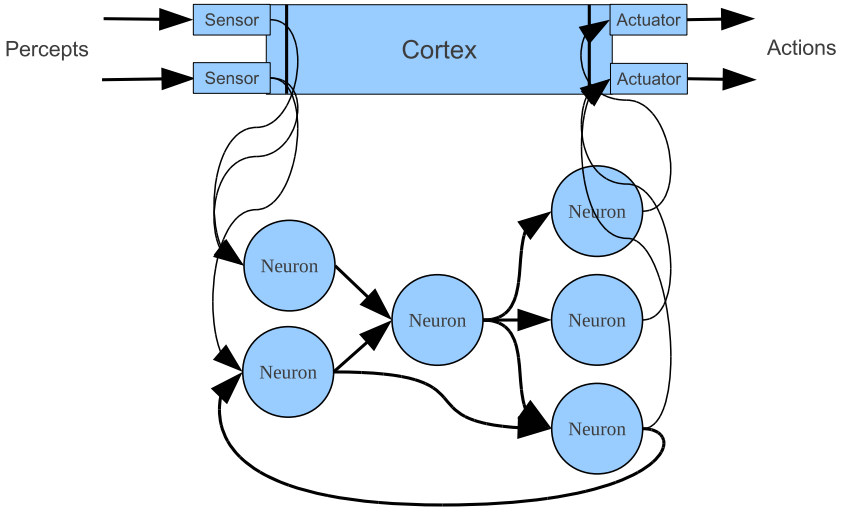

Fig. 10.1 The original DXNN based NN agent architecture.

The way a NN based system shown in Fig-10.1 functions, is as follows:

1. The genotype is first converted to the phenotype, composed of the cortex and the neurons, with the above shown architecture.
2. The cortex goes through all the sensor function names in its sensors list which has the following format*: [{Sensor1,[{N_Id1,FilterTag1}, {N_Id2, FilterTag2}...]}...]*. The cortex executes the sensor function names, and aggregates and packages the sensory signals generated through execution of the sensor functions. Because in the sensor list each sensor function comes with a list of neuron ids to which the resulting sensory signals are destined for, it is able to fanout those sensory signals to the specified neurons. Furthermore, in the above shown sensor list, the FilterTag has the following format: {single,Index}. {block,VL}, and {all,VL}. These tuples specify whether the sensor with a sensory signal of size vl, sends the entire sensory signal to the neuron, or just a single value from that vector list, a value located in the vector list at some particular Index, respectively. The third FilterTag: {all, VL}, specifies that the cortex will append the sensory signals of all the sensors, and forward that list to the neuron in question.
3. The cortex then gets the neuron ids stored in the *ct* list, and forwards sensory signals to them, by mapping from the ct neuron ids to the sensor list neuron ids and their corresponding sensory vector signals (this design made sense when I was originally building the system, primarily because it originally also supported supervised learning, which required this design).
4. The neurons in the NN then process the sensory signals until the signals are generated by the neurons in the output layer.
5. The output layer neurons send their results to the cortex.

6. The cortex, as soon as it sends all the sensory signals to the neurons, waits until it receives the signals from the neurons whose PIds are the same as the PIds in its *cf* list, which are the signals destined for the actuators. It gathers these signals into its accumulator, which is a list of lists, since the incoming signals are vectors.
7. After having gathered all the signals from the neurons, the cortex uses the actuators list and maps the composed output signal vectors to their respective actuators, and then executes the actuator functions using the output vectors as parameters.
8. **GOTO 2**

The original DXNN uses this particular convoluted architecture because I have developed it over a number of years, adding on new features, and modifying old features. Rather than redesigning the system once I've found a better way to represent or implement something, I simply modified it. DXNN has a modular version as well [2], where the evolved NN system is composed of modules called cores, where each core is a neural circuit, as shown in Fig-10.2. In the modular DXNN, the cores can be hopfield networks, standard evolved neural networks, and even substrate encoded NNs. At one point, long ago, DXNN even had a back-propagation learning mode, which I eventually removed as I never used it, and it was inferior to the non supervised learning algorithms I created. It is this long history of development, trial and error, testing and benchmarking, that left a lot of baggage in its architecture and implementation. Yet it is functional, and performs excellently.

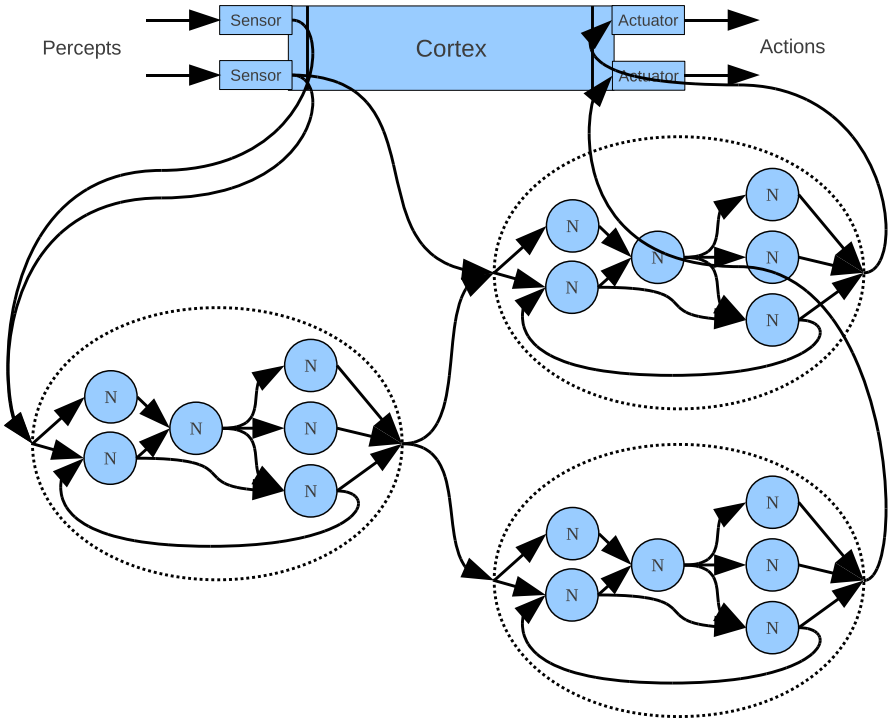

Fig. 10.2 Modular DXNN.

In some sense, the neural modules within the modular DXNN system, were meant to be used in emulation of the various brain regions. In this manner I hoped to evolve different regions independently, and then put them together into a complete system, or evolve the different modules at the same time as a single NN system, or even let the NN start of as monolithic, and then modularize through evolution. The performance though could not be established to be superior to standard homogeneous NN version at the time of experimentation, due to not yet having found a project benefiting from such an architecture. Nevertheless, the lessons learned were invaluable. The architecture of the TWEANN system we are developing in this book, is made with future use of modules in mind. Indeed, the system we are developing here will not only have more features, and will be more decoupled, but also its architecture will be cleaner, its implementation easier to understand, read, and expand, than that of DXNN. In the following sections I will explain the functionality, algorithms, and features that DXNN possesses.

10.3 Generational Evolution

I will first provide a simple list based overview of the steps taken by DXNN's general neuroevolutionary algorithm, and then elaborate on each of the more complicated sub-algorithms the DXNN system uses. When using the generational approach, DXNN uses the following set of steps:

1. **Initialization Phase:**
   Create a seed population of size K of topologically minimalistic NN genotypes.
2. **DO (Generational Neuroevolutionary loop):**
   3. Convert genotypes to phenotypes.
   4. **DO (Tuning Phase):**
      5. Test fitness of the NN system.
      6. Perturb recently added or mutation operator affected synaptic weights.

   UNTIL: NN's fitness has not increased for M times in a row.
   7. Convert the NN systems back to their genotypes, with the now updated and tuned synaptic weights.
   8. **Selection Phase:**
      9. Calculate the average energy cost of each neuron using the following method:
      TotFitnessPoints = Agent_Fitness(1) + Agent_Fitness(2) + ...Agent_Fitness(K),
      TotPopNeurons = Agent_TotNs(1) + Agent_TotNs(2) + ...Agent_TotNs(K),
      AvgNeuronCost = TotFitnessPoints/TotPopNeurons.

10. With all the NNs having now been given their fitness score, sort the genotypes based on their scores.
11. Mark the top 50% of the population as valid (fit), and the bottom 50% of the population as invalid (unfit).
12. Remove the bottom 50% of the population.

13. Calculate # of offspring for each agent:

14. For every agent(i) in K, calculate:
Agent(i)_NeuronsAllotted=Agent_Fitness(i)/AvgNeuronCost,
Agent(i)_OffspringAlloted=
Agent(i)_NeuronsAlloted/Agent(i)_TotNs

15. To keep the population size of the new generation the same as the previous, calculate the population normalizer, and then normalize each agent's allotted offspring value:
TotNewOffspring = Agent(1)_OffspringAlloted +
...Agent(i)_OffspringAlloted
Normalizer = TotNewOffspring/(K/2)

16. Now calculate the normalized number of offspring alloted for each agent:
Agent(i)_OffspringAllotedNorm =
round(Agent(i)_OffspringAlloted/Normalizer)

17. Create *Agent(i)_OffspringAllotedNorm* number of clones for every *Agent(i)* that belongs to the fit subset of the agents in the population. And then send each clone through the topological mutation phase, which converts that clone into an offspring.

18. Topological mutation phase:

19. Create the offspring by first cloning the parent, and then applying to the clone, T number of mutation operators. The value *T* is randomly chosen with uniform distribution to be between 1 and *sqrt(Agent(i)_TotNeurons),* where *TotNeurons* is the number of neurons in the parent NN. Thus, larger NNs will produce offspring which have a chance of being produced through a larger number of applied mutation operators.
20. Compose the population of the next generation by combining the genotypes of the fit parents with their newly created offspring.

UNTIL: Termination condition is reached (max # of evaluations, time, or fitness goal)

A diagram of this algorithm is shown in Fig-10.3. The steps 1 (Initialization phase), 4 (Parametric Tuning Phase), 8 & 13 (The Selection Phase & Offspring Allocation), and 18 (Topological Mutation Phase), are further elaborated on in the subsections that follow.

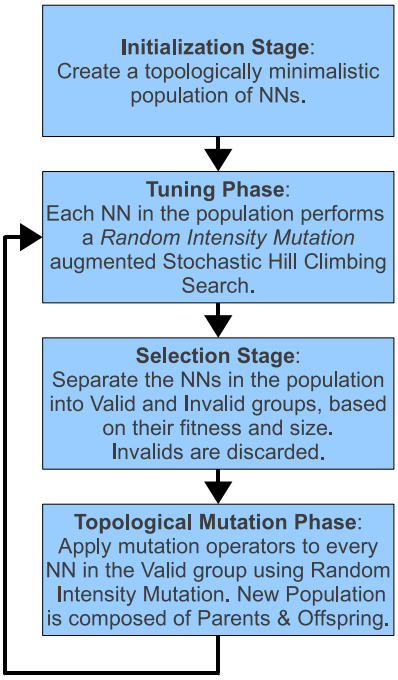

Fig. 10.3 The different stages in the DXNN's learning algorithm: Initialization Stage, Tuning Phase, Selection Stage, and Topological Mutation Phase.

10.3.1 Step-1: Initialization Phase

During the initialization, every element created has its Generation set to 0. Initially a seed population of size X is created. Each agent in the population starts with a minimal network, where the minimal starting topology depends on the total number of Sensors and Actuators the researcher decides to start the system with. If the NN is set to start with only 1 Sensor and 1 Actuator with a *vl = 1*, then the DXNN starts with a single Cortex containing a single Neuron. For example, if the output is a vector of length 1 like in the Double Pole Balancing (DPB) control problem, the NN is composed of a single Neuron. If on the other hand the agent is initiated with N number of Sensors and K number of actuators, the seed NNs will contain 2 layers of fully interconnected Neurons. The first layer contains S Neurons, and the second contains A_1+...A_k Neurons. In this topology, S is the total number of Sensors, and A_i is the size of the vector that is destined for Actuator *i*. It is customary for the NNs to be initialized with a single Sensor and a single Actuator, letting the agents discover any other auxiliary Sensors and Actuators through topological evolution.

Furthermore, the link from a Cortex to a Neuron can be of 3 types listed below:

1. Single-type link, in which the Cortex sends the Neuron a single value from one of its Sensors.
2. Block-type link, in which the Cortex sends the Neuron an entire vector that is output by one of the Sensors.
3. All-type link, in which the Cortex sends the Neuron a concatenated list of vectors from all the Sensors in its SensorList.

All this information is kept in the Cortex, the Neuron neither knows what type nor originally from which sensor the signal is coming. Each neuron only keeps track of the list of nodes it is connected from and the vector lengths coming from those nodes. Thus, to the Neuron all 3 of the previous link-types look exactly the same in its InputList, represented by a simple tuple {From_Id, Vector_Length}. The Vector_Length variable might of course be different for each of those connections.

The different link-types add to the flexibility of the system and allow the Neurons to evolve a connection where they can concentrate on processing a single value or an entire vector coming from a Sensor, depending on the problem's need. I think this improves the general diversity of the population, allows for greater compactness to be evolved, and also improves the NN's ability to move through the fitness landscape. Since it is never known ahead of time what sensory values are needed and how they need to be processed to produce a proper output, different types of links should be allowed.

For example, a Cortex is routing to the Neurons a vector of length 100 from one of its Sensors. Assume that a solution requires that a Neuron needs to concentrate on the 53rd value in the vector and pass it through a cosine activation function. To do this, the Neuron would need to evolve weights equaling to 0 for all other 99 values in the vector. This is a difficult task since zeroing each weight will take multiple attempts, and during random weight perturbations zeroing one weight might un-zero another. On the other hand evolving a single link-type to that Sensor has a 1/100 chance of being connected to the 53rd value, a much better chance. Now assume that a solution requires for a neuron to have a connection to all of the 100 values in the vector. That is almost impossible to achieve, and would require at least 100 topological mutations if only a single link-type is used, but has a 1/3 chance of occurrence if we have *block*, *all*, and *single* type links at our disposal. Thus the use of Link-Types allows the system to more readily deal with the different and wide ranging lengths of signal vectors coming from the Sensors, and having a better chance of establishing a proper connection needed by the problem in question.

In a population, the agents themselves can also be of different types: Type = "neural", and Type = "substrate". The "neural" type agent is one that is a standard recursive Neural Network system. The "substrate" type agents use an architecture where the NNs drive a neural substrate, an encoding that was popularized by HyperNEAT [3]. In such agents the sensory vector is routed to the substrate and the output vector that comes from the substrate is parsed and routed to the actua-

tors. The supervised NN itself is polled to produce the weights for the embedded neurodes in the substrate. The type of substrates can further differ in density, and dimensionality. A diagram of the agent architecture that utilizes a substrate encoding is shown in Fig-10.4. We will discuss the substrate encoded NN systems in greater detail in section 10.5.

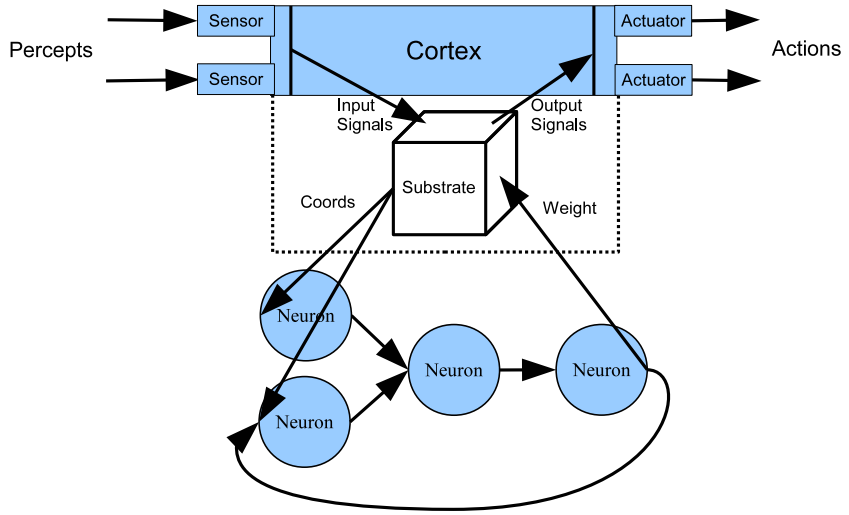

Fig. 10.4 A DXNN evolved agent that uses a substrate encoded based architecture. In this figure the cortex goes through its sensors to produce the sensory signals, which it then packages and passes to the Substrate, which produces output signals and passes those to the Cortex which then postprocesses them and executes its actuators using these output vectors as parameters. The Substrate uses the NN to set the weights of its embedded neurodes.

10.3.2 Step-4: Parametric Tuning Phase

Since the offspring is created by taking the fit parent, creating its clone, and then applying topological mutation operators to it, we can tag any neuron in the NN that has been affected by the mutation operator. What counts as been affected by the mutation operator is as follows:

1. Having been just created, for example when a new neuron has just been added to the NN.
2. Having just acquired new input or output connection, for example when a neuron has just created a new link to another element, or when another element has just created a link to the neuron in question, the neuron is counted as having been affected by the mutation operator.
3. When during the topological mutation phase, the neuron's activation function, plasticity, or another parameter (other than weights) has been mutated.

Instead of just giving to such neurons the "mutationally affected" tag, their generation parameter is reset, the same as is the case in the system we've built thus far. Thus, every element in the NN is given a generation during the initial seed population creation, and then every time the element is affected by a mutation, its generation is reset to the current generation, where the "current" generation is N where N increments every topological mutation phase, and is kept track of by the agent element. In this manner we can track which parts of the NN have been mutating, and which topological structures have stabilized and for a number of generations have not been affected by mutation. This stabilization usually occurs when the mutation of such structures produces a less fit offspring than its parent. So we can then, using this approach pick out the stabilized structures and *crystallize* them, making those structures a single unit (and be potentially represented by a single process) that in the future will no longer be disturbed by mutation.

To choose whose synaptic weights to perturb during the tuning event, first the exoself chooses a random generation limit value as follows: *GenLimit = 1/random:uniform()* where the random:uniform() function generates a random value between 0 and 1 with a uniform distribution. Thus GenLimit will always be greater than 1, and have 50% of being 2, 25% of being 4... DXNN then uses the randomly generated GenLimit to compose a pool of neurons which have been affected by mutations within the last GenLimit of generations. In this neuron pool each neuron is chosen with a probability of *1/sqrt(NeuronPoolSize)* to have its synaptic weights perturbed. The list of these chosen neurons is called the New Generation Neurons (NGN). The chosen neurons are then each sent a message by the *exoself* to have their synaptic weights perturbed. When a neuron receives such a message, it goes through its synaptic weight list and chooses each weight for perturbation with a probability of *1/sqrt(TotSynapticWeights)*. The neuron then perturbs the chosen synaptic weights with a value randomly generated with uniform distribution between -Pi and Pi.

This particular approach has the following benefits: 1. It concentrates on tuning and optimizing neurons that have only recently been added to the NN, thus ensuring that newly added neurons can contribute in a positive way to the NN. 2. There is a high variability in the number of neurons and weights that are chosen at any given time, thus there are times when a large number of neurons are all perturbed at the same time, and there are times when, by chance alone, only a few neurons and a few of their synaptic weights are chosen. Thus this approach allows the system to have a chance of doing both, tune into local optima on the fitness landscape, and also at times choose a large number of neurons and weights to perturb, and thus search far and wide in the parametric space.

After NGN is composed, a variable *MaxMistakes* is created and set to *abs*(*BaseMaxMistakes + sqrt(TotWeights from NGNs))* rounded to the nearest integer. The *BaseMaxMistakes* variable is set by the researcher. Finally, a variable by the name *AttemptCounter* is created and set to 1.

The reason for the creation of the NGN list is due to the weight perturbations being applied only to the these new or recently modified Neurons, a method I refer to as "*Targeted Tuning*". The reason to only apply perturbations to the NGNs is because evolution in the natural world works primarily through complexification and elaboration, there is no time to re-perturb all the neurons in the network after some minor topological or other type of addition to the system. As NNs grow in size it becomes harder and harder to set all the weights and parameters of all the Neurons at the same time to such values that produces a fit individual. A system composed of thousands of neurons might have millions of parameters in it. The odds of finding proper values for them all at the same time by randomly perturbing synaptic weights throughout the entire system after some minor topological mutation, is slim to none. The problem only becomes more intractable as the number of Neurons continues to grow. By concentrating on tuning only the newly created or newly topologically/structurally augmented Neurons and making them work with an already existing, tuned, and functional Neural Network, makes the problem much more tractable. Indeed in many respects it is how complexification and elaboration works in the biological NNs. In our organic brains the relatively recent evolutionary addition of the Neocortex was *not* done through some refurbishing of an older NN structure, but through a completely new addition of neural tissue covering and working with the more primordial parts. The Neocortex works concurrently with the older regions, contributing and when possible overwriting the signals coming from our more ancient neural structures evolved earlier in our evolutionary history.

During the Tuning Phase each NN based agent tries to solve the problem based on its morphology. Afterwards, the agents receive fitness scores based on their performance in that problem. After being scored, each NN temporarily backs up its parameters. Every neuron in the NGN list has a probability of *1/sqrt(Tot_NGNs)* of being chosen for weight perturbation. The Exoself sends these randomly chosen neurons a request to perturb some of their weights. Each chosen Neuron, after receiving such a message, chooses a set of its own synaptic weights, and perturbs them. The total number of weights to be perturbed is chosen randomly by every Neuron itself. The number of weights chosen for perturbation by each neuron is a random value between 1 and square root of total number of weights in that Neuron. The perturbation value is chosen with uniform distribution to be between -(WeightLimit/2) and (WeightLimit/2), where the WeightLimit is set to 2*Pi. By randomly selecting the total number of Neurons, the total number of weights to perturb, and using such a wide range for the perturbation intensity, we can achieve a very wide range of parametric perturbation. Sometimes the NN might have only a single weight in a single Neuron perturbed slightly, while at other times it might have multiple Neurons with multiple weights perturbed to a great degree. This allows the DXNN platform to make small intensity perturbations to fine tune the parameters, but also sometimes very large intensity (number of Neurons and weights) perturbations to allow NN based agents to jump over or out of local optima, an impossibility when using only small perturbations applied

to a small number of Neurons. This high mutation variability method is referred to in the DXNN platform as the *Random Intensity Mutation* (RIM). The range of mutation intensities grows as the square root of the total number of NGNs, as it logically should since the greater the number of new or recently augmented Neurons in the NN, the greater the number of perturbations that needs to be applied to make a significant effect on the information processing capabilities of the system. At the same time, the number of neurons and weights affected during perturbation is limited only to the newly/recently added or topologically augmented elements, so that the system can try to adjust the newly added structures and those elements that are directly affected by them through new connections, to work and positively contribute to an already existing neural system.

After all the weight perturbations have been applied to the NN based agent, it attempts to solve the problem again. If the new fitness achieved by the agent is greater than the previous fitness it achieved, then the new weights overwrite the old backed up weights, the AttemptCounter is reset to 1, and a new set of weight perturbations is applied to the NN based agent. Alternatively, if the new fitness is not greater than the previous fitness, then the old weights are restored, the AttemptCounter is incremented, and another set of weight perturbations is applied to the individual.

When the agent's *AttemptCounter* == *MaxMistakes*, implying that a MaxMistakes number of unsuccessful RIMs have been applied in sequence without a single one producing an increase in fitness, the agent with its final best fitness and the correlated weights is backed up to the database through its conversion back to a list of tuples, its genotype, followed by the termination of the agent itself. Utilizing the AttemptCounter and MaxMistakes strategy allows us, to some degree at least, test each topology with varying weights and thus let each NN after the tuning phase to represent roughly the best fitness that its topology can achieve. In this way there is no need to forcefully and artificially speciate and protect the various topologies since each NN represents roughly the highest potential that its topology can reach in a reasonable amount of time after the tuning phase completes. This allows us to judge each NN based purely on its fitness. If one increases the *BaseMaxMistakes* parameter, then on average each NN will have more testing done on it with regards to weight perturbations, thus testing the particular topology more thoroughly before giving it its final fitness score. On the other hand the MaxMistakes parameter itself grows in proportion to the square root of the total sum of NGN weights that should be tunned, since the greater the number of new weights that need to be tuned, the more attempts it would take to properly test the various permutations of neurons and their synaptic weights.

10.3.3 Step-8 & 13: The Selection & Offspring Allocation Phase

There are many TWEANNs that implement speciation during selection. Speciation is used to promote diversity and protect unfit individuals who in the current generation do not possess enough fitness to get a chance of producing offspring or mutating and achieving better results in the future. Promoters of speciation algorithms state that new ideas need time to develop and speciation protects such innovations. Though I agree with the sentiment of giving ideas time to develop, I must point to [4] in which it was shown that such artificial and forced speciation and protection of unfit organisms can easily lead to neural bloating. DXNN platform does not implement forced speciation, instead it tests its individuals during the Tuning Phase and utilizes natural selection that also takes into account the complexity of each NN during the Selection Stage. In my system, as in the natural world, smaller organisms require less energy and material to reproduce than their larger counterparts. As an example, for the same amount of material and energy that is required for a human to produce and raise an offspring, millions of ants can produce and raise offspring. When calculating who survives and how many offspring to allocate to each survivor, the DXNN platform takes complexity into account instead of blindly and artificially defending the unfit and insufficiently tested Neural Networks. In a way, it can also be thought that every NN topology represents a specie in its own right, and the tuning phase concisely tests out the different parametric permutations of that particular specie, same topologies with different weights. I believe that speciation and niching should be done not forcefully from the outside by the researcher, but by the artificial organisms themselves within the artificial environments they inhabit, if their environments/problems allow for such a feat. When the organisms find their niches, they will automatically acquire higher fitness and secure their survival that way.

Due to the Tuning Phase, by the time Selection Stage starts, each individual presents its topology in roughly the best light it can reach within reasonable time. This is due to the consistent application of Parametric RIM to each NN during targeted tuning, and that only after a substantial number of continues failures to improve is the agent considered to be somewhere at the limits of its potential. Thus each NN can be judged purely by its fitness rather than have a need for artificial protection. When individuals are artificially protected within the population, more and more Neurons are added to the NN unnecessarily, thus producing the dreaded neural/topological bloating. This is especially the case when new neurons are added, yet the synaptic weight perturbation and mutation is applied indiscriminately to all the synaptic weights in the NN. Topological bloating dramatically and catastrophically hinders any further improvements due to a greater number of Neurons unnecessarily being in the NN and needing to have their parameters set *concurrently* to just the right values to get the whole system functional. An example of such topological bloating was demonstrated in the robot arm control experiment using NEAT and EANT2 [4]. In that experiment, NEAT continued to fail due to significant neural bloating, whereas EANT2 was successful, which like DXNN is

a memetic algorithm based TWEANN. Once a NN passes some topological bloating point, it simply cannot generate enough of concurrent perturbations to fix the faulty parameters of all the new neurons it acquired. At the same time, most TWEANN algorithms allow for only a small number of perturbations to be applied at any one instance. In DXNN, through the use of *Targeted Tuning* and *RIMs* applied during the Tuning and Topological Mutation phases, we can successfully avoid bloating.

Finally, when all NNs have been given their fitness rating, we must use some method to choose those NNs that will be used for offspring creation. DXNN platform uses a selection algorithm I call "Competition", which tries to take into account not just the fitness of each NN, but also the NN's size. The c*ompetition* selection algorithm is composed of the following steps:

1. Calculate the average energy cost of the Neuron using the following steps:
   TotEnergy = Agent(1)_Fitness + Agent(2)_Fitness...
   TotNeurons = Agent(1)_TotNeurons + Agent(2)_TotNeurons...
   AverageEnergyCost = TotEnergy/TotNeurons
2. Sort the NNs in the population based on their fitness. If 2 or more NNs have the same fitness, they are then sorted further based on size, more compact solutions are considered of higher fitness than less compact solutions.
3. Remove the bottom 50% of the population.
4. Calculate the number of alloted offspring for each Agent(i):
   AllotedNeurons = (Fitness/AverageEnergyCost),
   AllotedOffsprings(i) = round(AllotedNeurons(i)/Agent(i)_TotNeurons)
5. Calculate total number of offspring being produced for the next generation:
   TotalNewOffsprings = AllotedOffsprings(1)+...AllotedOffsprings(n).
6. Calculate PopulationNormalizer, to keep the population within a certain limit:
   PopulationNormalizer = TotalNewOffsprings/PopulationLimit
7. Calculate the normalized number of offspring alloted to each Agent:
   NormalizedAllotedOffsprings(i) =
   round(AllotedOffsprings(i)/PopulationNormalizer(i)).
8. If NormalizedAllotedOffsprings (NAO) == 1, then the Agent is allowed to survive to the next generation without offspring, if NAO > 1, then the Agent is allowed to produce (NAO -1) number of mutated copies of itself, if NAO = 0 the Agent is removed from the population and deleted.
9. The Topological Mutation Phase is initiated, and the mutator program then passes through the database creating the appropriate NAO number of mutated clones of the surviving agents.

From this algorithm it can be noted that it becomes very difficult for bloated NNs to survive when smaller systems produce better or similar results. Yet when a large NN produces significantly better results justifying its complexity, it can begin to compete and push out the smaller NNs. This selection algorithm takes into account that a NN composed of 2 Neurons is doubling the size of a 1 Neuron NN, and thus should bring with it sizable fitness gains if it wants to produce just

as many offspring. On the other hand, a NN of size 101 is only slightly larger than a NN of size 100, and thus should pay only slightly more per offspring. This is exactly the principle behind the "competition" selection algorithm we implemented in the system we are developing together in this book.

10.3.4 Step-18: The Topological Mutation Phase

An offspring of an agent is produced by first creating a clone of the parent agent, then giving it a new unique Id, and then finally applying Mutation Operators to it. The *Mutation Operators* (MOs) that operate on the individual's topology are randomly chosen with uniform distribution from the following list:

1. "Add Neuron" to the NN and link it randomly to and from randomly chosen Neurons within the NN, or one of the Sensors/Actuators.
2. "Add Link" (can be recurrent) to or from a Neuron, Sensor, or Actuator.
3. "Splice Neuron" such that that two random Neurons which are connected to each other are disconnected and reconnected through a newly created Neuron.
4. "Change Activation Function" of a random Neuron.
5. "Change Learning Method" of a random Neuron.
6. "Add Bias", all neurons are initially created without bias.
7. "Remove Bias", removes a bias value in the neurons which have one.
8. "Add Sensor Tag" which connects a currently unused Sensor present in the SensorList to a random Neuron in the NN. This mutation operator is selected with a researcher defined probability of X. In this manner new connections can be made to the newly added or previously unused sensors, thus expanding the sensory system of the NN.
9. "Add Actuator Tag" which connects a currently unused Actuator present in the ActuatorList to a random Neuron in the NN. This mutation operator is selected with a researcher defined probability of Y. In this manner new connections can be made to the newly added or previously unused actuators, thus expanding the types of tools or morphological properties that are available for control by the NN.

The "Add Sensor Tag" and "Add Actuator Tag" can both allow for new links from/to the Sensor and Actuator programs not previously used by the NN to become available to it. In this manner the NN can expand its senses and control over new actuators and body parts. This feature becomes especially important when the DXNN platform is applied to the Artificial Life and Robotics experiments where new tools, sensors, and actuators might become available over time. The different sensors can also simply represent various features of a problem, and in this manner the DXNN platform naturally incorporates feature selection capabilities.

The total number of Mutation Operators (MOs) applied to each offspring of the DXNN is a value randomly chosen between 1 and square root of the total number of Neurons in the parent NN. In this way, once again a type of random intensity

mutation (RIM) approach is utilized. Some mutant clones will only slightly differ from their NN parent, while others might have a very large number of MOs applied to them, and thus differ drastically. This gives the offspring a chance to jump out of large local optima that would otherwise prove impassible if a constant number of mutational operators were to have been applied every time, independent of the parent NN's complexity and size. As the complexity and size of each NN increases, each new topological mutation plays a smaller and smaller part in changing the network's behavior, thus a larger and larger number of mutations needs to be applied to produce significant differences to the processing capabilities of that individual. For example, when the size of the NN is a single neuron, adding another one has a large impact on the processing capabilities of that NN. On the other hand, when the original size is a million neurons, adding the same single neuron to the network might not produce the same amount of change in the computational capabilities of that system. Increasing the number of MOs applied based on the size of the parent NN's size, allows us to make the mutation intensity significant enough to allow the mutant offspring to continue producing innovations in its behavior when compared to its parent, and thus exploring the topological fitness landscape far and wide. At the same time, due to RIM, some offspring will only acquire a few mutations and differ topologically only slightly and thus have a chance to tune and explore the local topological areas on the topological fitness landscape.

Because the sensors and actuators are represented by simple lists of existing sensor and actuator programs, just like in the system we're developing together in this book, the DXNN platform allows for the individuals within the population to expand their affecting and sensing capabilities. Such abilities integrated naturally into the NN lets individuals gather new abilities and control over functions as they evolve. For example, originally a population of very simple individuals with only distance sensors is created. At some point a fit NN will create a mutant offspring to whom the "Add Sensor Tag" or "Add Actuator Tag" mutational operator is applied. When either of these mutational operators is randomly applied to one of the offspring of the NN, that offspring then has a chance of randomly linking from or to a new Sensor or Actuator respectively. In this manner the offspring can acquire color, sonar or other types of sensors present in the sensor list, or acquire control of a new body part/actuator, and thus further expand its own morphology. These types of expansions and experiments can be undertaken in the artificial life/robotics simulation environments like the Player/Stage/Gazebo Project [5]. Player/Stage/Gazebo in particular has a list of existing sensor and actuator types, making such experiments accessible at a very low cost.

Once all the offspring are generated, they and their parents once more enter the tuning phase to continue the cycle as was diagrammed in Fig-10.3.

10.4 Steady-State Evolution

Though the generational evolution algorithm is the most common approach, when applying neuroevolutionary systems to ALife, or even non ALife simulations and problems, steady-state evolution offered by DXNN can provide an advantage due to its content drift tracking ability, and a sub population called "Dead Pool" which can immediately be used to develop committee machines. In a steady-state evolution, the population solving the problem or existing within the simulated world (in the case of ALife for example) always maintains a constant operational population. When an organism/agent dies, or when there is more room in the environment (either due to the expansion of the food source in ALife environment, or because more computational power is added, or more exploration is wanted...) more concurrently existing agents are added to the operational phenotypes. The system does not wait for every agent in the population to finish being evaluated before generating a new agent and entering it into the population. Instead, the system computes the fitness of the just having perished agent, and then immediately generates a new genotype from a pool of previously evaluated fit genotypes. Thus the system maintains a relatively constant population size by consistently generating new offspring at the same pace that agents complete their evaluations and are removed from the live population.

In DXNN, the steady-state evolutionary algorithm uses an "Augmented Competition" (AC) selection algorithm. The AC selection algorithm keeps a list of size "PopulationSize" of dead NN genotypes, this list is called the "dead pool". The variable *PopulationSize* is specified by the researcher. When an Agent dies, its genotype and fitness is entered into this list. If after entering the new genotype into the dead pool the list's size becomes greater than PopulationSize, then the lowest scoring DXNN genotype in the dead pool is removed. In this manner the dead pool is always composed of the top performing PopulationSize number of ancestor genotypes.

In this augmented version of the selection algorithm, the AllottedOffspring variables are converted into normalized probabilities used to select a parent from the dead pool to produce a mutated offspring. Finally, there is a 10% chance that instead of creating an offspring, the parent itself will enter the environment/scape or be re-applied to the problem. Using this "re-entry" system, if the environment or the manner in which the fitness is allotted changes, the old strategies and their high fitness scores can be re-evaluated in the changed environment to see if they still deserve to stay in the dead pool, and if so, what their new fitness should be. This selection algorithm also has the side effect of having the dead pool implicitly track content drift of the problem to which the TWEANN is applied.

For example assume that the steady-state evolution with the *dead_pool* list is applied to an ALife simulation. Every time an agent in the simulated environment dies (has been evaluated), it is entered into the dead pool, and a new offspring is generated from the best in this dead pool. Once the dead pool size reaches that of

PopulationSize specified by the researcher, the DXNN system also begins to get rid of the poorly performing genotypes in the dead_pool. But what is important is that when an organism in the environment dies, there is a chance that a genotype in the dead pool has a chance of re-entering the simulated environment, instead of a new mutant offspring being generated. If it were not for this, then as the environment changes with the dynamics and fitness scoring and life expectancy all changing with it... and some organism dies, the old organisms, the genotypes from the "old world" would be used to create the offspring. If the environment is highly dynamic and malleable, after a while the whole thing might change, the useful survival instincts and capabilities that were present in the environment to which the dead_pool organisms belonged, might no longer be present in the current, evolved environment. Suddenly we would be faced with a dead_pool of agents all with high scores, which though achievable in the previously simple environment, are no longer possible in the now much more complex and unforgiving environment. Thus it is essential to re-evaluate the organisms in the dead_pool, are they still fit in the new environment, in the environment that itself has evolved and become more complex? Can the old agents compete in the new world?

The re-entry system allows us to change and update the dead_pool with the organisms that are not simply more fit, but are more fit in the current state of the environment. The environment can be either the simulated environment of the ALife system, or the new signal block in the time series of currency-pairs or stock prices for example. The patterns of the market that existed last year, might have changed completely this year, and it is essential that the new agents are judged by how they perform on this year's patterns and styles of the time series. This is the benefit of the content drift tracking dead pool. The dead pool represents the best of the population, a composition of agent genotypes that perform well in the relatively new environment, that perform well in the world of today, rather than the one of last year.

Furthermore, because the genotypes belonging to the dead_pool represent the best of the population, we can directly use the genotypes in it to compose a committee machine. The current state of the dead pool is the voting population of the committee machine, the type of system we discussed in Section-1.2.2. This type of setup is shown in Fig-10.5.

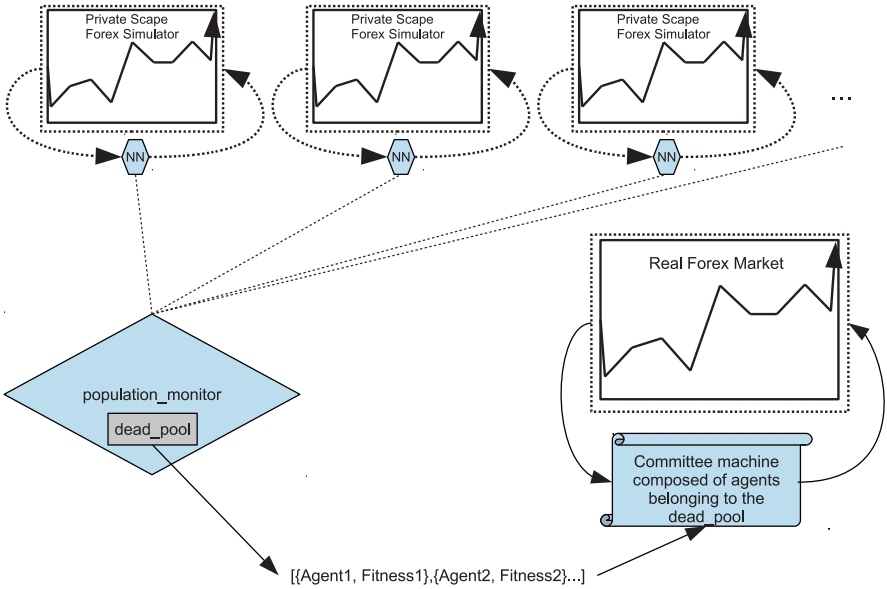

Fig. 10.5 A DXNN system using steady-state evolution used to evolve currency trading agents, and whose dead pool is used as a committee machine applied to real Forex trading.

The steps of the steady-state evolution algorithm in the DXNN platform are as follows:

1. **Initialization Phase**:
    2. Create a seed population of size K of topologically minimalistic NN genotypes.
    3. Convert genotypes to phenotypes.
4. **DO (Steady-State Neuroevolutionary loop)**:
    5. **For Each Agent, DO (Tuning Phase)**:
        6. Test fitness of the NN system
        7. Perturb the synaptic weights of recently added or mutation operator affected neurons

    UNTIL: NN's fitness has not increased for X times in a series

    8. Convert the NN system back to its genotype, with the now updated and tuned synaptic weights.
    9. Add the agent's genotype to the dead_pool list of size K.
    10. **Steady-State Selection Phase (For genotypes in the dead_pool list)**:
        11. Calculate the average energy cost of each neuron using the following method:
        TotFitnessPoints = Agent_Fitness(1) + Agent_Fitness(2) + ...Agent_Fitness(K),
        TotPopNeurons = Agent_TotNs(1) + Agent_TotNs(2) + ...Agent_TotNs(K),
        AvgNeuronCost = TotFitnessPoints/TotPopNeurons.

12. With all the NNs having now been given their fitness score, sort the genotypes based on their scores.
13. Extract the top K agents in this sorted dead_pool list, delete the others. This is done for the case when the addition of the new agent to the dead_pool, makes the size of the dead_pool larger than K. We only want to keep K agents in the dead_pool.
14. **Select a dead_pool champion agent**:
    15. Agent(i)_NeuronsAllotted = Agent_Fitness(i)/AvgNeuronCost, Agent(i)_OffspringAllotted = Agent(i)_NeuronsAllotted/Agent(i)_TotNs
    16. Convert Agent(i)_OffspringAllotted for each agent into a normalized percentage, such that a random agent from this list can be chosen with the uniform distribution probability proportional to its Agent(i)_OffspringAllotted value.
    17. Choose the agent through step-16, and designate that agent as dead_pool champion.
    18. Randomly choose whether to use the dead_pool champion as the parent of a new offspring agent, or whether to extract the champion from the dead_pool, convert it to its phenotype, and re-apply it to the problem. The split is 90/10, with 90% chance of using the champion's genotype to create a new offspring, and 10% chance of removing the agent from the dead_pool and re-applying (aka re-entry, re-evaluation...) the agent to the problem.
19. **IF champion selected to create offspring**:
    20. **Topological mutation phase**:
        21. Create the offspring by first cloning the parent, and then applying to the clone T number of mutation operators, T is randomly chosen to be between 1 and sqrt(Agent(i)_TotNeurons). Where the TotNeurons is the number of neurons in the parent NN, and T is chosen with uniform distribution. Thus larger NNs will produce offspring which have a chance of being produced through a larger number of applied mutation operators to them.
    22. Designate the offspring agent as **New_Agent**.

ELSE champion is chosen for re-entry:

    23. Extract agent from the dead_pool.
    24. Designate the agent as **New_Agent**.

25.Convert the agent designated as **New_Agent** to its phenotype.

UNTIL: Termination condition is reached (max # of evaluations, time, or fitness goal)

As can be noted from these steps, the algorithm is similar to the generational evolutionary approach, but in this case as soon as an agent dies (if in ALife experiment), or finishes its training or being applied to the problem, its fitness is immediately evaluated against the dead_pool agents, and a new agent is created (either through offspring creation or re-entry) and applied to the problem, or released into the simulated environment.

The tuning phase and the topological mutation phase are the same as in the generational evolutionary loop, discussed in the previous section. The steady-state selection algorithm only differs in that the allotted_offspring value is converted to a percentage of being selected for each agent in the dead_pool. The selected agent has a 90% chance of creating an offspring and 10% chance of being sent back to the problem, and being re-evaluated with regards to its fitness.

The following sections will cover a few finer points and features of DXNN. In the next section we will discuss its two types of encoding, neural and substrate. In section 10.6 we will briefly discuss the flatland simulator, a 2d ALife environment. In section 10.7 we will discuss the modular version of DXNN. Finally, in section 10.8 and 10.9 we will discuss the ongoing projects and features being integrated into the DXNN system, and the neural network research repository being worked on by the DXNN Research Group.

10.5 Direct (Neural) and Indirect (Substrate) Encoding

The DXNN platform evolves both direct and indirect encoded NN agents. The direct encoded NN systems are as discussed in the above sections, these are standard neural networks where every neuron is encoded as a tuple, and the mapping from the genotype to phenotype is direct. We simply translate the tuple containing the synaptic weights and link specifications into a process, linked to other processes and possessing the properties and synaptic weights dictated by the tuple.

The indirect encoding that the DXNN can also use is a form of substrate encoding, popularized by the HyperNEAT [3]. There are many variations of substrate encoding, and new ones are turning up every year. In a substrate encoded NN, the actual NN is not directly used to process input sensory signals and produce output signals to control the actuators. Instead, in a substrate encoded NN system the NN "paints" the synaptic weights and connectivity patterns on a multidimensional substrate of interconnected neurodes. This substrate, based on the synaptic weights determined by the NN, is then used to process the input sensory signals and pro-

duce output signals used by the actuators. The architecture of such a system is shown in the following figure.

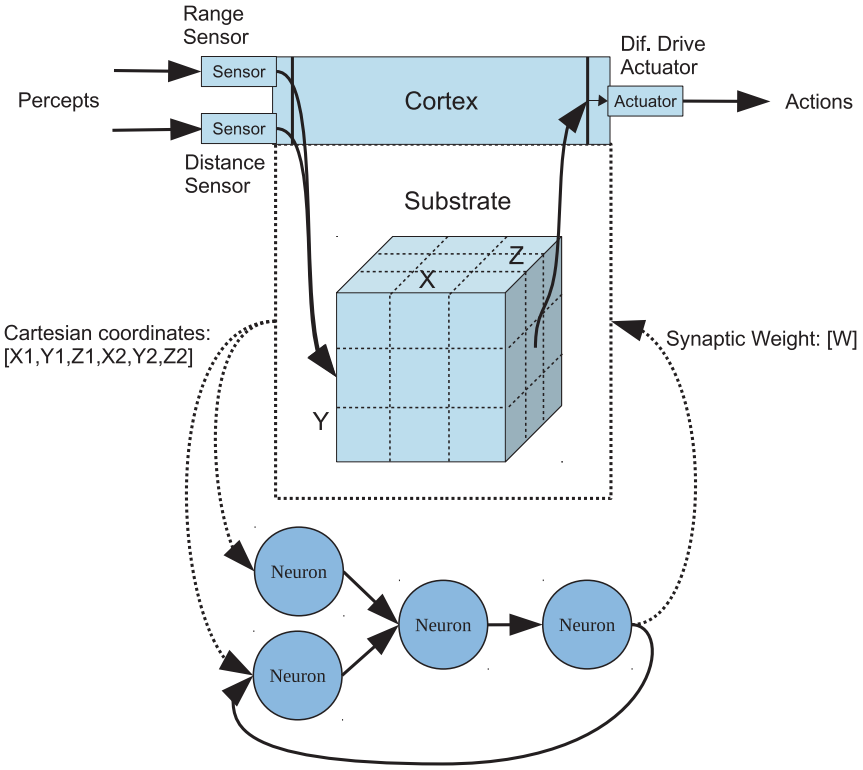

Fig. 10.6 Substrate encoded neural network system. This diagram is of a substrate encoded agent. The substrate, sensors, and actuators, are all part of the same process called Cortex. The NN is used to generate the synaptic weights between neurodes in the substrate, based on the coordinates of the presynaptic and postsynaptic neurodes. The sample agent shown is one that controls a simulated robot in an ALife experiment, a simulated robot that has a Range Sensor, a Distance Sensor, and a Differential Drive Actuator.

The neurodes in the substrate all use the sigmoid or tanh activation function, though this of course can be changed. Furthermore, the NN's output can be used for anything, and not only used as the synaptic weights for the coordinate specified neurodes. For example, the output of the NN can be used and considered as the *Delta Weight*, the change in the synaptic weight between the pre- and post-synaptic neurodes, based on the coordinates of the said neurodes fed to the NN, in addition with the pre-synaptic neurode's output, the post-synaptic neurode's output, and the current synaptic weight between the two. We will further discuss the details of substrates and their functionality in the following section, followed by a discussion of the genotype encoding DXNN uses for substrates, the phenotype representation that it uses for such substrate encoded agents, and finally the different types of "substrate_sensors" and "substrate_actuators", which further modify

the substrate encoded NN systems, allowing the NN to not only use the coordinates of the two connected neurodes when computing the synaptic weight between them, but various other geometrically significant features, like distance, spherical coordinates, planner coordinates, centripetal distance...

10.5.1 Neural Substrates

A neural substrate is simply a hypercube structure whose axis run from -1 to 1 on every dimension. The substrate has neurodes embedded in it, where each neurode has a coordinate based on its location within the hypercube. The neurodes are connected to each other, either in the feed forward fashion, a fully connected fashion, or random connection based fashion. An example of a 2d substrate is shown in Fig-10.7a, and a 3d substrate in Fig-10.7b.

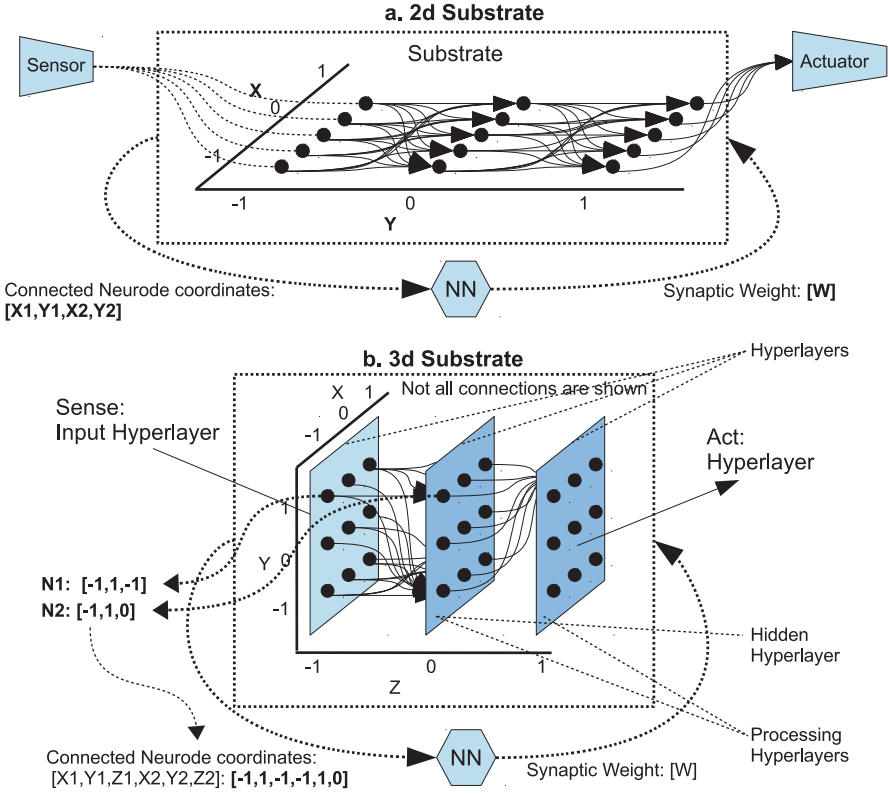

Fig. 10.7 An example of different substrates in which the neurodes are connected to each other in the feed forward fashion.

The density of the substrate refers to the number of neurons on a particular axis. For example, if the substrate is a 2d one, and the density of the substrate is 5 by 3, then this plane substrate has 5 neurons, uniformly distributed on the x axis, with 3 total of such layers, which too are uniformly distributed on the y axis, as shown in Fig-10.7a. The Fig-10.7b shows a 3d substrate with the density distribution of 3x3x3. In this substrate, there are 3 planes on the Z axis, where each plane is composed out of 3x3 neurode patterns. Each plane is connected to the plane ahead of it, hence it is a feed forward based substrate, since the signals travel from the *-Z* direction, towards the *+Z* direction. We could of course have a fully connected substrate, where every neurode is connected to every other neurode. Also, the substrate does not necessarily need to be symmetric, it can have any type of pattern, any number of neurons per layer or hyperlayer, and positioned in any pattern within that layer or hyperlayer.

From these examples you can see that the processing, input, and output hyperlayers, are one dimension lower than the entire substrate. The sensory signals travel from the negative side of the axis of the most external dimension (Y in the case of 2d, and Z in the case of 3d in the above examples), from the input hyperlayer, through the processing hyperlayers, and finally to the output hyperlayer, whose neurodes' output counts as the output of the substrate (but again, we could designate any neurode in the substrate as an output neurode, and wait until all such output neurodes produce a signal, and count that as the substrate's output). The manner in which we package the output signals of the neurodes within the output hyperlayer, and the manner in which we feed those packaged vectors to the actuators, determines what the substrate encoded NN based agent does. Finally, because the very first hyperlayer is the input to the substrate, and the very last hyperlayer is the output of the substrate, there must be at least 2 hyperlayers making up the substrate structure.

For example, assume we'd like to feed an image coming from a camera into the substrate encoded NN system. The image itself is a bitmap, let's say of resolution 10x10. This is perfect, for this type of input signal we can create a 3d substrate with a 10x10 input hyperlayer, 3x3 hidden processing hyperlayer, and a 1x5 output hyperlayer. Each hyperlayer is a 2d plane, all positioned on the 3rd dimension, thus making the substrate 3d, as shown in Fig-10.8. As can be seen, the input being the very first layer located at Z = -1, has its signals sent to the second layer, located at Z = 0, which processes it, and whose neurode outputs are sent to the 3rd layer at Z = 1, processed by the last 5 neurodes whose output is considered the final output of the substrate.

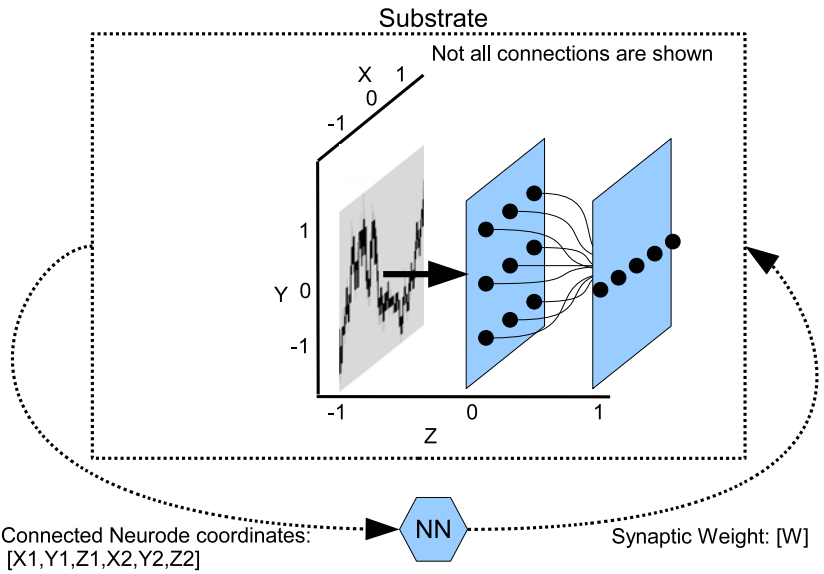

Fig. 10.8 A [{10,10},{3,3},{5,1}] substrate being fed a 2d plane image with a 10x10 ({10,10}) resolution.

You might be asking at this point "What is the advantage of using substrate encoding?" The answer is in the way we produce their weights. The weights are determined by the NN which calculates the synaptic weight between two connected neurodes based on their coordinates. The coordinates of the connected neurodes act as the input to the NN. Since the NN has the coordinates as input, it can do the following:

1. Extract geometrical patterns in the input hyperlayer, and thus it can be applied to highly complex problems where such geometrical information can be exploited.
2. Be used to generate weights for very large and very dense substrates, with the connectivity and synaptic weight patterns based on the coordinates, and thus being of almost any complexity and form.
3. Due to never seeing the actual input signals, it cannot evolve a single synaptic weight for some particular element in the input vector during training, it cannot evolve some specific set of synaptic weights to pick out a particular single small pattern. In other words, a substrate encoded NN has a much lower chance of overtraining. It paints the synaptic weights broadly on the substrate, and thus it should be able to generalize that much better.
4. Because the NN produces a smooth function, and because each neurode in the substrate has presynaptic connections from a smooth spread of neurodes, with regards to their coordinates in the previous hyperlayer, the synaptic weights produced by the NN for any particular neurode, varies smoothly. This is the reason why it is much more difficult for such synaptic weights to overtrain on

some single particular points in the input stream of signals. Hence the superior generalization. The NN paints the synaptic weights and connectivity patterns on the substrate in "broad strokes", so to speak.

Let us discuss some of the things mentioned in more detail.

Geometrical Feature Sensitivity:

As discussed, the input to the NN is a list of coordinates for the connected pre-synaptic and post-synaptic neurodes. Not only are the coordinates used as input to the NN, but also the coordinates can be first converted to spherical coordinates, polar coordinates, distance between the connected neurodes, distance to the center of the substrate... before they are fed to the NN. Because a NN is a universal function approximator, and the inputs are various geometrical elements, and because the input hyperlayer itself has coordinates, the NN gains the ability to pick out and deal with the geometrical features of the substrate, and the sensory signals.

Large Neural Network Structures:

Since the substrate neurode density is independent of the actual NN which we evolve, through substrate encoding it is possible to create very large/dense substrates, with thousands or millions of neurodes. Thinking again about the substrate analyzing the data/images coming from a camera, we can also see that the denser the substrate, the higher the resolution of images it can analyze. Also the resolution of the sensory inputs and the output of the substrate, are independent of the NN painting the connectivity and synaptic weights on it. The "curse of dimensionality" does not plague this type of system as much, since we can concentrate on a smaller number of evolving parameters and topologies (of the actual evolving NN), while controlling a vast substrate embedded NN. Finally, it is also possible to implement synaptic plasticity using *iterative*, *abc,* and other types of substrate learning rules [6], which we will discuss in detail and implement in later chapters.

The "Broad Stroke" property:

Because the neural network that calculates the synaptic weights for the neurodes in the substrate does not see the actual input vectors, and instead only deals with the coordinates. And because the output of the NN is a smooth function, and the input coordinates to the NN are based on the connected neurodes, and each neurode is connected from a whole spectrum of neurodes in the previous hyper-layer, with their coordinates changing smoothly from -1 to 1. The synaptic weights are painted in "Broad Strokes". Meaning, due to the inability of the NN to pick out any particular points in the incoming data, the synaptic weights it generates are smooth over the whole substrate. A change in the NN system changes the weights, the output function of the substrate, in general *and smoothly*, bringing values smoothly up or down... This means that over-training is more difficult because the weights of the neurodes do not lock up on some single particular data point in the input signals. Thus the generalization of the substrate encoded agent is superior, as was shown in papers: "Evolving a Single Scalable Controller for

an Octopus Arm with a Variable Number of Segments" [7] and "Evolving Chart Pattern Sensitive Neural Network Based Forex Trading Agents" [12].

10.5.2 Genotype Representation

As we saw in Fig-10.7, the substrate is part of the cortex process. The genotypical specification for the cortex element in DXNN is:

```
{id,sensors,actuators,cf,ct, type,plasticity, pattern,cids,su_id, link_form,dimensions, densities, generation}
```

This tuple specifies the substrate dimensionsionality and its general properties through the *dimensions* and *densities* elements. Because the sensors and actuators of the substrate are independent of the actual substrate itself, the neurode densities of the substrate, the specification for the "processing hyperlayers", the "input hyperlayers", and the "output hyperlayers", are independent. Though this may at first sound somewhat convoluted, after the explanation you will notice the advantages of this setup, especially for a neural network based system that is meant to evolve and grow.

When I say "processing hyperlayer" I mean the substrate hyperlayer (2d, 3d… substrate layer of neurodes) that actually has neurodes that process signals. As was noted in the discussion on the substrate, the sensory inputs, which are sometimes multidimensional like in the case of the signals coming from a camera, are part of the substrate, located at the *-1* side of the axis defining the depth of the substrate. The output hyperlayers of the substrate are of the processing type. Because the input hyperlayers and output hyperlayers need to be tailored for the particular set of sensors and actuators used by the agent, the input hyperlayers, processing hyperlayers, and the output hyperlayers of the substrate, are all specified separately from one another.

So, to create the initial substrate for the agent, the substrate's topology is specified in 3 parts. First DXNN figures out how many dimensions the substrate will be composed of. This is done by analyzing all the sensors and actuators available to the agent. In DXNN, the sensors and actuators not only specify the vector lengths of the signals, but also the geometrical properties (if any) that the signals will exhibit. This means that they specify whether the input signals are best viewed or analyzed as a plane with a resolution of X by Y, or a cube of a resolution X by Y by Z, or if there is no geometrical data and that the vector length L of the input signal can be viewed as just a list. If the NN based agent is substrate based, then the DXNN platform will use this extra geometry specification information to create the substrate topology most appropriate for it. Thus, if the morphology of the seed population being created is composed of 2 sensors and 3 actuators as follows:

```
sensors:
[#sensor{name=distance_scanner,id=cell_id,format={symmetric,Dim}, tot_vl=pow(Res,Dim),
parameters=[Spread,Res,ROffset]} || Spread<-[Pi/2],Res<-[5], Roffset<-[Pi*0/2]] ++
[#sensor{name=color_scanner,id=cell_id,format={symmetric,Dim}, tot_vl=pow(Res,Dim),
parameters=[Spread,Res,ROffset]} || Spread <-[Pi/2], Res <-[4], Roffset<-[Pi*0/2], Dim=2],

actuators:
[#actuator{name=two_wheels,id=cell_id,format=no_geo,tot_vl=2,parameters=[]},
#actuator{name=create_offspring,id=cell_id,format=no_geo,tot_vl=1,parameters=[]},
#actuator{name=spear,id=cell_id,format=no_geo,tot_vl=1,parameters=[]}]
```

Where the *parameters* element specifies the extra information necessary for the proper use of the sensor or actuator, and the *format* element specifies the geometrical formatting of the signal. We can see that the actuators all have their formats set to *no_geo* meaning, no geometric information, so the actuators expect from the substrate single dimensional vector outputs. On the other hand, the sensors both use format= {symmetric,2}, which specifies a two dimensional sensory signal with a symmetric resolution in both dimensions: X by Y where X = Y. The parameters also specify, since these sensors are part of the simulated robot with distance and color sensors, the simulated sensor's coverage area (Spread), camera resolution (Res), and sensor's radial offset from the robot's central line (based on the actual simulation of the robot which is specified during the ALife simulation). Based on the format, the DXNN knows that the sensors will produce two symmetric 2d input signals, with a resolution of 5 and 4 respectively. Thus the first sensory input will be a 5x5 plane, and the second a 4x4 plane. The DXNN also knows that the actuators expect single dimensional output vectors, the one called *two_wheels* expects the signal sent to it be a vector of length 2, with the other two actuators expecting the signals sent to them to also be single dimensional lists, vectors, and in this case of length 1 (the length is specified by the tot_vl parameter).

Having this information, DXNN knows to expect input signals that will be at least 2d (new sensors might be added in the future, which might of course have higher, or lower dimension), and that the output signals will be 1d. The DXNN thus calculates that the input hyperlayer composed of multiple 2d inputs will be at least 3d (2d planes stacked on a 3rd dimension), and the output hyperlayer will be at least 2d (1d outputs stacked on the 2nd dimension), which means that the substrate must be at least 4d. But why 4d?

Though certainly it is possible to devise substrates whose dimension is the same as the highest dimension of the sensor or actuator used by it, I usually implement a layer to layer feedforward substrate topology which requires the substrate's dimension to be the maximum sensor or actuator dimension, *+2*. The reasoning for this is best explained through an example.

Let's say the substrate encoded NN based agent uses 2 sensors, each of which is 2d, and 2 actuators, each of which is 1d. The 2d input planes do not perform any type of processing because the processing is done in the hidden processing hyperlayers, and the output hyperlayer. So both of the 2d input planes must forward their signals to the processing hyperlayers. So we must first put these 2d planes on another dimension. Thus, to form an input hyperlayer we first put the 2d planes on the third dimension, forming a 3d input hyperlayer. But for the 3d input hyperlayer to forward its signals to another 3d processing hyperlayer, we need to put both on a 4th dimension. Thus, the final substrate is 4d. The input hyperlayer is 3d. The output hyperlayer, though really only needing to be 2d (due to the output signals being both 1d layers stacked on a 2nd dimension to form an output hyperlayer), is also 3d because all neurodes haves to have the same dimensionality.

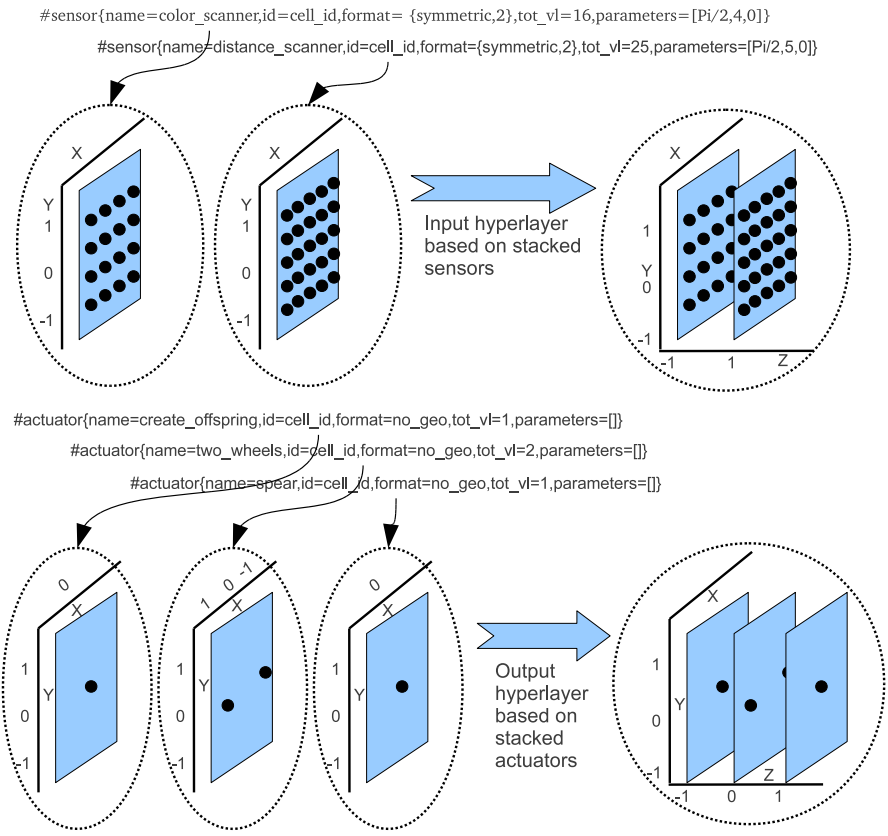

Fig. 10.9 Input and output hyperlayers composed by stacking the sensor input planes into a single multidimensional input hyperlayer, and stacking the output processing planes into a single multidimensional output hyperlayer with signals destined for actuators.

Why give an extra dimension to put the input or output planes on? Because in the future we might want to add more sensors and actuators, and have the sensors and actuators stacked on another dimension makes it easy to do so. For example we would simply add the new sensor based input plane on the same 3rd dimension, and scoot the others a bit. In this manner we can add new sensors and actuators indefinitely, without changing the substrate topology too much. Also the coordinates of the neurodes in the input planes would change only slightly due to scooting, and so the synaptic weights determined by the NN could be more easily and smoothly adjusted through synaptic weight tuning phase.

This is the gist of the idea when forming substrates dynamically, based on sensors and actuators used, and expecting to use multiple such sensors and actuators in the future. We will discuss substrate encoding in much more detail in Chapter-16.

So, now we know how to compute the dimensionality of the input and output hyperlayers. The number of the processing hyperlayers, if any (in the case where only the input and output hyperlayers exist) is determined by the depth value set by the researcher. In DXNN, the hidden processing hyperlayers, their topology and dimensionality, is set to the resolution equal to the square root of the highest resolution between the sensors and actuators of the agent's morphology.

Thus through this process, when creating the seed population of the substrate encoded NN based agents, DXNN can calculate both the dimension of the substrate to create, its topology, and the resolution of each dimension. The resolution of each hidden processing hyperlayer is set to square root of the highest resolution of the signals coming from the sensors or towards actuators. The dimensionality is set, as noted earlier, to the highest dimension between the sensors and actuators, +2. The depth, the number of total hidden processing hyperlayers, is set by the researcher, usually to 0 or 1. If it is set to 0, then there is only the input and output (which is able to process the sensory signals) hyperlayers, and 0 hidden processing hyperlayers. When set to 1, the full substrate is composed of the input hyperlayer, the hidden processing hyperlayer whose resolution was computed earlier from the resolution of the sensors and actuators, and the processing output hyperlayer whose dimensionality and topology was formed by analyzing the list of available actuators for the agent, and the list of the actuators currently used by the agent.

For example, the substrate created based on the morphology composed from the following sensors and actuators:

sensors:
[#sensor{name=internals_sensor,id=cell_id,format=no_geo,tot_vl=3,parameters=[]},
#sensor{name=color_scanner,id=cell_id,format={symmetric,2}, tot_vl=Density,
parameters=[Spread,Res,ROffset]} || Spread <-[Pi/2], Res <-[4], ROffset<-[Pi*0/2]],

actuators:
[#actuator{name=two_wheels,id=cell_id,format=no_geo,tot_vl=2,parameters=[]},
#actuator{name=create_offspring,id=cell_id,format=no_geo,tot_vl=1,parameters=[]}]

Will have the hidden processing hyperlayer resolutions set to 2, the dimensionality set to 4 = **2**+2, and the depth set by the researcher to 1. Since the input and output hyperplanes are created when the genotype is converted to phenotype, and based on the number and types of sensors involved, assuming that in this example the agent is using all the sensors and actuators, the substrate will have the following form:

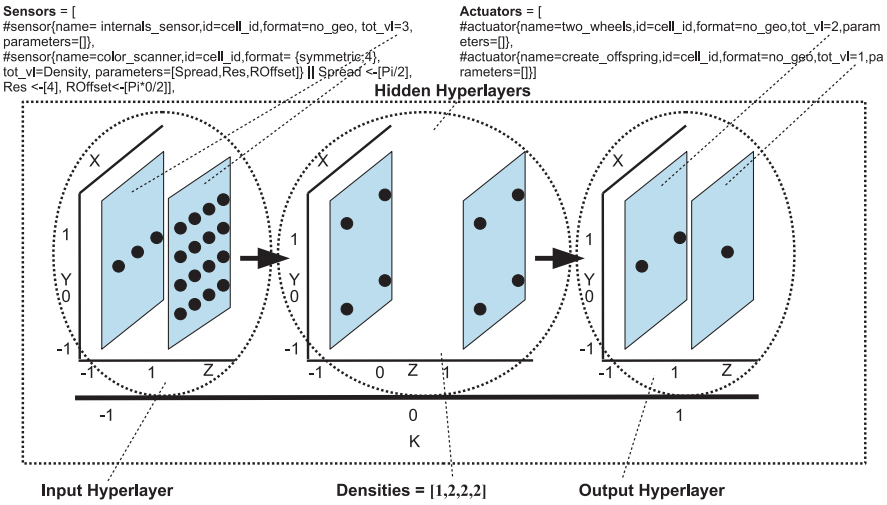

Fig. 10.10 The substrate belonging to an agent with 2 sensors and 2 actuators, with the dimensions = 4, and densities = [1,2,2,2]

Once the substrate and its properties are determined, the actual NN is then created in a fashion similar to one created when standard direct/neural encoding is used. Only in a substrate encoded NN based system, the sensors and actuators

used by the NN are the substrate_sensors and substrate_actuators, because it is the substrate that is using the real sensors and actuators, while the NN gets its input (coordinates and other neurode parameters) from the substrate, and uses its output signals to execute the substrate_actuators, which set up the synaptic weights (and other parameters) between the neurodes.

In the NNs that use substrate encoding, since it is the substrate that accepts inputs from the environment and outputs signals to the outside world, and the NN is just used to set the weights for the neurodes in the substrate, the system not only has a set of sensors and actuators as in the standard NN system, but also a set of substrate_sensors and substrate_actuators. The substrate_sensors and substrate_actuators are used by the NN in the same way the standard, neural encoded NN uses sensors and actuators, and new substrate_sensors and substrate_actuators are also in the same way integrated into the NN as it evolves.

In the standard substrate encoded NN system, the NN is given an input that is a vector composed of the coordinates of the two neurodes that are connected. In DXNN, the set of substrate_sensors are coordinate processors that process the coordinate vectors before feeding the resulting vector signals to the NN. The substrate_actuators on the other hand process the NN's output, and then based on their function interact with the substrate by either setting the neurode synaptic weights, changes a neurode's currently set synaptic weights (which effectively adds plasticity to the substrate), or performs some other function.

The DXNN system currently has the following list of substrate_sensors available for the substrate encoded NNs:

1. *none:* Passes the Cartesian coordinates of the connected neurodes directly to the NN.
2. *cartesian_distance:* Calculates the Cartesian distance between the neurodes, and passes the result to the NN.
3. *polar_coordinates* (if substrate is 2d): Transforms the Cartesian coordinate vector to the polar coordinate vector, and passes that to the NN.
4. *spherical_coordinates* (if substrate is 3d): Transforms the Cartesian coordinate vector to the spherical coordinate vector, and passes that to the NN.
5. *centripetal_distance*: Transforms the Cartesian coordinate vector from the connected neurodes into a vector of length 2, composed of the distances of the two neurodes to the center of the substrate.
6. *distance_between_neurodes*: Calculates the distance between the two connected neurodes, and passes that to the NN.

This set of substrate_sensors further allows the substrate encoded NN to extract the geometrical patterns and information from its inputs, whatever dimension those input signals have. An example of an architecture of a substrate encoded NN using multiple substrate_sensors and multiple substrate_actuators, with the substrate itself using multiple sensors and actuators as well, is shown in Fig-10.11.

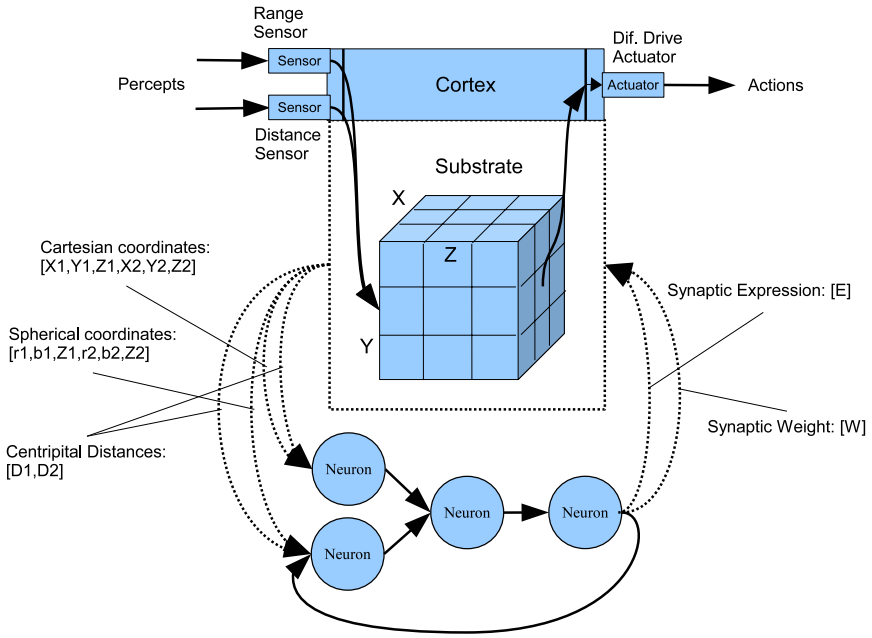

Fig. 10.11 A substrate encoded NN using different types of substrate_sensors and substrate_actuators, and standard sensors and actuators.

As can be seen from the figure, it is also possible to have different types of substrate_actuators, not just the standard *synaptic_weight* substrate_actuator which uses the NN's output to set the synaptic weight between the two neurodes based on their coordinates which were passed to the NN's substrate_sensors. The standard substrate_actuator, *synaptic_weight* setter, is one that simply uses the signal coming from the NN and converts it into a synaptic weight value using the algorithm shown in Listing-10.1. In this listing, the substrate_actuator simply takes the NN's output, and computes the synaptic weight to be 0 if the NN's output is between -0.33 and 0.33, and to be between -1 and 1 otherwise, normalizing the synaptic weight value such that there is no hole between -0.33 and 0.33 when using this below shown function:

Listing-10.1 The simple synaptic weight setting "substrate_actuator".

```
set_weight(Output)->
    [Weight] = Output,
    Threshold = 0.33,
    Processed_Weight = if
            Weight > Threshold ->
```

```
                (functions:scale(Weight,1,Threshold)+1)/2;
        Weight < -Threshold ->
                (functions:scale(Weight,-Threshold,-1)-1)/2;
        true ->
                0
end.
```

Currently there are a number of other substrate_actuators implemented as well. For example a secondary substrate_actuator called *synaptic_expression,* decides on whether there is a connection between the two neurodes at all, if there isn't then the weight is set to 0. This is different from the weight being set to 0 by the synaptic_weight actuator, since using this secondary actuator the whole substrate can be made more complex, there can be two different neural circuits, one deciding on the synaptic weight, and one deciding on the connectivity pattern of the substrate. Or for example instead of using the *synaptic_weight* actuator, an *iterative_plasticity*, *abc_plasticity,* or some other learning algorithm can be used. Using these plasticity substrate_actuators, the NN can change and modify the synaptic weights after every sensory input is processed. One substrate_actuator could be mutated into another during the topological mutation phase, new ones could be added or removed throughout evolution.

These substrate_actuators further allow one to experiment with different types of learning, adding more agility and robustness to the population and individual agents, providing a greater leverage to evolution to overcome various discontinuities and abstractions on the fitness landscape. Combined all together, with the various substrate specific mutation operators which increase the resolution/density of the substrate, add new sensors and actuators, add new substrate_sensors and substrate_actuators... the substrate encoding provided by the DXNN system is one of the most advanced substrate encoded neuroevolutionary approaches currently available.

The resolution and dimensionality of the substrate can be further mutated during the topological mutation stage. When the agent is substrate encoded, the platform's standard mutation operator list is further augmented to include the following substrate specific mutation operators:

1. mutate_resolution
2. mutate_dimensionality

Yes the method and representation is convoluted and could be made simpler. The problem with DXNN, as noted earlier, is that it was built up slowly, evolving through many of my various experiments and tests. And as we know, evolution does not take the cleanest path from genotype A to genotype Z, instead it is all based on the easiest and most direct path, which is based on the agent's environment, and most easily achievable niche based on the agent's genotype/phenotype at that time. Here too, the DXNN is the way it is because of the order in which I got the ideas for its various parts, and the initial, though at times mistaken, repre-

sentations and implementations I used. Once a few hundred or thousand lines of code are written, the amount of motivation to recreate the system in a cleaner manner decreases. But now that we are creating a completely new TWEANN system together, and have the knowledge of my earlier experience within the field and systems like DXNN to guide us, we can create our new system with foresight, without having to go down the same dark alleys and dead ends I wondered into during my first time around.

10.5.3 Substrate Phenotype Representation

The conversion of genotype to phenotype is similar to one used by the standard direct encoded NNs in DXNN, and thus is similar to what we use in the system we've built so far. As we discussed, in DXNN the cortex process is not a synchronizer but instead is the signal gatekeeper between the NN and the sensors and actuators it itself is composed of. In the substrate encoded NNs, the cortex also takes on the role of the substrate itself. In DXNN, the entire *[substrate, cortex, sensors, actuators, substrate_sensors, substrate_actuators]* system is represented as a single element/process, because it is possible to encode the substrate in a list form and very efficiently perform calculations even when that substrate is composed of thousands of neurodes.

When the exoself generates and connects all the elements (neurons and the cortex), it does so in the same way it does with the direct encoded NN system. Since the cortex knows, based on its parameters, that it is a substrate encoded system, once it is created it builds a substrate based on dimension, densities, sensor, actuator, substrate_sensor, and substrate_actuator list specifications. The neurons and the NN that they compose neither know nor need to know that the agent is substrate encoded. In both versions, the direct encoded and the indirect encoded NN system, the input and the output layer neurons are connected to the cortex, so nothing changes for them. The cortex is the one that needs to keep track of when to use the substrate sensors/actuators, and when to use the actual sensors/actuators.

The algorithm that the substrate encoded cortex follows is specified in the following steps, with a follow-up paragraphs elaborating on the more intricate parts.

1. The cortex process is spawned by the exoself, and immediately begins to wait for its initial parameters from the same.
2. The cortex receives all its *InitState* in the form:

```
{ExoSelf_PId,Id,Sensors,Actuators,CF,CT,Max_Attempts,Smoothness,OpMode,Type,
Plasticity, Morphology,Specie_Id,TotNeurons,Dimensions,Densities}
```

3. The cortex checks the agent Type, whether it is *neural* or *substrate.* In the steps that follow we assume that the Type is *substrate*.

4. **Cortex constructs the substrate:**
    5. The cortex reads the number of dimensions, and the densities.
    6. The input hyperlayer is built based on the sensors the agent uses, with the neurode coordinates based on the number of dimensions (If the entire substrate has 3 dimensions, then each coordinate is [X,Y,Z], if 4d then [X,Y,Z,T]...).
    7. If *depth* > *0*, then hidden processing hyperlayers are constructed based on the densities and dimension specified, and with each neurode in the first hidden processing hyperlayer having the right number of synaptic weights to deal with the input hyperlayer.
    8. The output processing hyperlayer is constructed, and each neurode must have the right number of synaptic weights to deal with the signals coming from the hidden processing hyperlayers.
    9. The cortex combines the input, processing, and output hyperlayers into a single hypercube substrate.
10. **DO Sense-Think-Act loop:**
    11. **DO For each neurode in the substrate:**
        12. The cortex goes through the substrate_sensors, using the tuples like in the standard sensors, to forward the neurode properties (coordinates, and other parameters based on the substrate_sensor used) to the connected neurons in the NN.
        13. The output signals of the NN are then used to execute the substrate_actuators to set the synaptic weights and other parameters between the neurodes in the substrate.

    UNTIL: All neurodes have been assigned their synaptic weights and other parameters.

    14. The cortex goes through every sensor, and maps the sensory signals to the input hyperlayer of the substrate.
    15. The substrate processes the sensory signals.
    16. The output hyperlayer produces the output signals destined for the actuators. Since the output hyperlayer is created based on the actuators the agent uses, the output signals are implicitly of the right dimensionality and in the right order, such that the signals are packaged into vectors of proper lengths, and are then used as parameters to execute the actuator functions.
    17. The cortex goes through every actuator, executing the actuator function using the output signals produced by the substrate as the parameter of their respective actuators.

UNTIL: Termination condition is reached, tuning has ended, or interrupt signal is sent by the exoself.

During the tuning phase, after every evaluation of the NN, the exoself chooses which neurons should perturb their synaptic weights. After the neurons in the NN have perturbed their synaptic weights, the cortex takes the substrate through the

step 11 loop, updating all the synaptic weights of the neurodes in the substrate by polling the NN for weights.

Thus the cortex first executes all the sensor functions to gather all the sensory signals, then it goes through every neurode in the substrate, until the processing output hyperlayer produces the output signals, which the cortex gathers, packages into appropriate vectors, and executes all the actuators in its actuator list with the appropriate output vector signals.

The phenotypic architecture of the substrate encoded NN based agent, composed of the Exoself, Cortex, and Neuron elements, with the Sense-Think-Act loop steps specified, is shown in Fig-10.12.

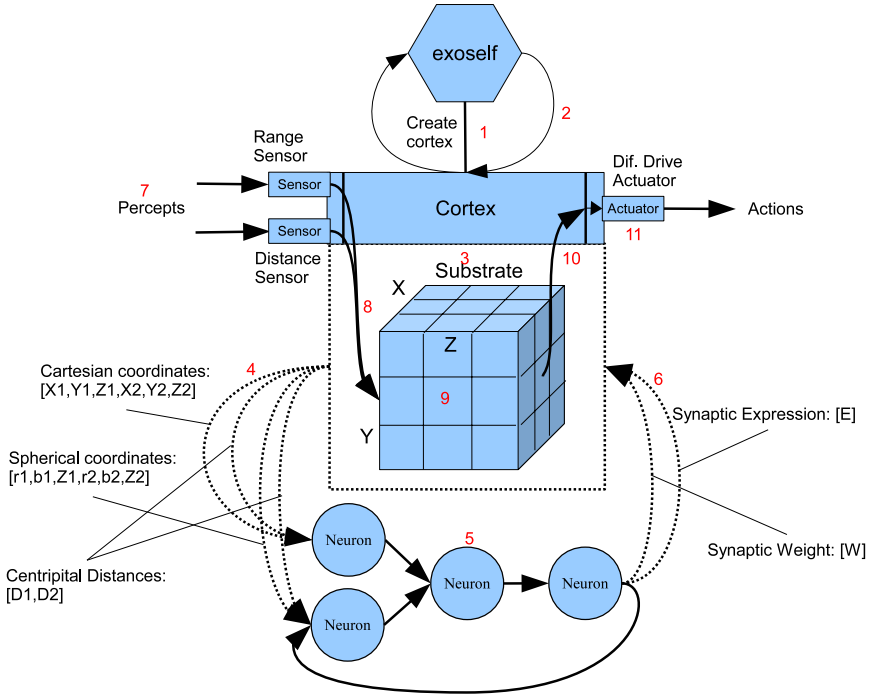

Fig. 10.12 The phenotypic architecture of the substrate encoded NN based agent, composed of the concurrent Exoself, Cortex, and Neuron processes, with the processing steps listed.

Let's quickly go over the shortened processing loop shown in the above figure.

1. The exoself creates the cortex.
2. The exoself sends the created cortex its *InitState* parameters.
3. The cortex creates the substrate based on sensors, actuators, and other specifications.
4. The cortex/substrate uses the substrate_sensors to forward to the NN the coordinates and other parameters of the connected neurodes within the substrate.
5. The NN processes the signals passed to it by its substrate_sensors.

6. The substrate_actuators and the signals produced by the NN, used as parameters for the substrate_actuators, are used to set the synaptic weights and other parameters of the embedded neurodes.
7. The cortex gathers the sensory signals from its sensors.
8. The cortex maps the sensory signals to the substrate's input hyperlayer.
9. The substrate processes the sensory signals coming from the input hyperlayer.
10. The cortex maps the output signals of the neurodes in the output hyperlayer to their appropriate actuators.
11. The cortex executes the actuators with the substrate produced output signals as their parameters.

The substrate, due to it being a single process, and capable of being composed of millions of neurodes each with millions of connections, and because each neurode simply does vector multiplication, is a perfect candidate for being accelerated with a GPU. Substrate encoding is an important field of neurocomputation, it allows for very large NNs to be constructed, for neuroplasticity and geometrical pattern sensitive systems to be composed, and in general substrate encoded NNs are more effective, and perhaps with some new topological structure and with further expansions, might be the path toward general computational intelligence.

Have you ever seen a PET scan? You know that activity pattern that it shows? It is difficult not to look at the NN computing the synaptic weights and therefore activity pattern on the substrate, as the tool which could carve out that high density, and highly complex architecture. With a substrate having enough neurons (100 billion let's say), and with the NN, the universal function approximator, having the right function, it could possibly carve out the architecture and the activation patterns similar to something one would see in a PET scan... But we are not at that point just yet.

We will add substrate encoding capabilities to the TWEANN system we are developing together, and thus we will discuss further the algorithms and a way to represent the substrate in great detail. We will of course, having foresight, develop our system to have a more concise and flexible representation. As we develop the next generation TWEANN in this book, we will avoid making the mistakes I made when I first developed the architecture of DXNN.

In the following sections we will discuss the current and ongoing projects that DXNN is being used for, and thus what the system we're developing here (which will replace DXNN, by becoming the new DXNN) will be applied to once developed. The system we're creating here is meant to supersede and replace DXNN, it is the next generation of a fully concurrent, highly general and scalable, *Topology and Parameter Evolving Universal Learning Network* (TPEULN).

10.6 DXNN Research Group & the NN Research Repository

DXNN Research group [8] is currently a small research group composed of a few mathematicians, computer scientists, and me. We are working on further expanding the DXNN platform, and finding new projects to apply it to. One of these projects is the application of DXNN to Cyberwarfare. Another deals with exchanging the neuron elements with logic gates and transistors, so that the platform can be applied towards the evolution and optimization of large scale digital circuits. The currently explored application of DXNN is towards the evolution and optimization of OpenSPARC [9], some progress has been made, but not enough to publish. The DXNN Research group is also currently working on interfacing the DXNN with the *Player/Stage/Gazebo* [5] project, allowing it to be used in 3d ALife experiments, and the evolution of robotic systems and neurocontrollers for the same. The Player/Stage/Gazebo robot simulators provide 2d and 3d simulation environments, and the drivers to interface the evolved NNs with actual hardware. The use of *Player* gives us the ability to evolve systems in artificial environments, and immediately have the ability to apply them to real hardware, and thus usable and applicable in the real world. The current main project and interest in this area is the evolution of neurocontrollers for the control of Unmanned Combat Ariel Vehicles (UCAVs). This is accomplished through the co-evolution of two, forever warring, populations of Combat UAVs in the 3d simulated environment, through Gazebo for example. Due to the use of the Player interface, we can then transfer the evolved intelligence to real UCAV systems.

The main reasons why we are trying to create a highly decoupled neuroevolutionary system is because it will allow us to easily augment it, and then provide it to the public so that crowdsourcing is used to further expand the platform, letting anyone with interest and skill to contribute various modules and computational packages to the system, further expanding and augmenting it, making it more general, and applicable to new projects, which benefits the entire community using the TWEANN system. DXNN Neural Network Research Repository [10] provides the specifications on how to add new modules to the DXNN TWEANN, where to submit them...

The goal of the Neural Network Research Repository (NNRR) is also to become the repository of neural network systems evolved through the DXNN system. NNs are by their very nature blackbox systems, different neural networks can be evolved to solve the same problem, or inhabit same environments (when NN based agents are used in ALife). NNRR provides a place where individuals can submit the NN systems they have evolved, and specify the fitness functions and other parameters they used to evolve these agents. Because everyone else on the NNRR is also using DXNN, they can then try to see what types of NN topologies they can evolve given the same fitness function and TWEANN parameters. Thus, the NNRR should over time accumulate useful NN based agents. Those who wish to simply start using these agents can do so, others can try to download the hun-

dreds of the already evolved NN based systems for some problem, and try to datamine their topologies, try to see what are the essential parts of these NN based systems, what are the common threads? Through this approach we can try to start building a path towards illuminating the blackbox. These types of databases also provide the data needed to figure out where the DXNN system is perhaps having difficulties when solving problems.

Finally, with the standardized interfaces between the various processes, and with the specified genotypical encoding system, the community can contribute the various activation functions, neural plasticity rules, neuron types, substrate topologies, fitness functions, selection functions... Every decoupled element is a self contained module, and thus anyone can augment the DXNN system by simply conforming to the proper interface specifications. The NNRR will propel us, and allow for the capabilities and applicability of this neuroevolutionary system to expand dramatically, making the evolved systems available globally, providing already evolved solutions to those interested, and giving a place for researchers to contribute, while at the same time giving them a place where they can gather tools and data for their own further research.

10.7 Currently Active Projects

The DXNN research group is currently actively pursuing three projects:

1. Cyberwarfare.
2. Coevolution of Unmanned Ariel Vehicle Combat Maneuvers.
3. CPU Evolution and Optimization.

When successful, the results of these 3 projects could potentially be game changing for the industrial and military sector.

10.7.1 Cyberwarfare

One of the exciting applications the DXNN platform is currently being applied toward is the evolution of offensive and defensive cyberwarfare agents. We are currently trying to evolve agents capable of using parameterized metasploit (a penetration testing program) and other tools to effectively penetrate and attack other machines, and agents capable of defending their host network against such attacks, by monitoring signals on its network for attacks being carried out against it, and then using various available tools and methods to thwart and counterattack. This is done by creating scapes, simulated network environments using network simulators like NS3, with simulated host targets, and then interfacing the NN based agents with programs like metasploit, letting them evolve the ability to

combine the various attack commands to penetrate the simple hosts. With regards to the evolution of defensive agents, the NN based agents are fed signals coming from the ports, and they are required to make a decision of whether they are being actively attacked or not. If they are, they must decide on what they should do, lock the port, fully disconnect, counter-attack...

There are a number of difficulties in evolving cyberwarfare agents, because unlike in the natural environments, there are no smooth evolutionary paths from simply existing on a network, to being able to forge attack vectors using metasploit. Neither is there a smooth evolutionary path leading from mere existence, to the ability to detect more and more complex attacks being carried out against your own host. In standard ALife, there is an evolutionary path from simply running after a prey and then eating it, to trying different approaches, hiding, baiting the prey... it's all a smooth progression of intelligence. That is not the case in cyberwarfare, things are more disconnected, more arcane, requiring beforehand knowledge and experience. Nevertheless, through bootstrapping simple skills, and forging fitness functions, our preliminary results have demonstrated that the goals of this project are achievable.

10.7.2 Coevolving Unmanned Ariel Vehicle Combat Maneuvers

Another exciting application and field where evolved neurocognitive systems can provide a significant advantage is of course robotics. As with cyberwarfare, there is a significant amount of both industrial and military applications, with the successful system and implementation being potentially game changing. Due to the current increased use of unmanned aerial vehicles, particularly in combat, there is a great opportunity in evolving neural network agents specifically for controlling such systems. At the moment the UAVs are programmed to scout, or fly to particular way-points. Once the UAV gets there, a real pilot takes over. The pilot sits somewhere in the base and controls the UAV, looking at the screen which is fed by the UAV's camera. This of course provides a much lower level of situational awareness to the pilot when compared to that available when sitting in a cockpit. Also, the maneuvers available to the drone are limited by the human operator, and the time delay in the connection due to the distance of the UAV from the human operator. All of this combined, puts the Unmanned Combat Ariel Vehicle (UCAV) at a disadvantage in a standard dogfight against a piloted fighter jet. Yet a UCAV can undertake *g* forces and perform maneuvers that are impossible for a human piloted jet fighter. Furthermore, an evolved NN would be able to integrate the signals from many more sensors, and make the decisions faster, than any biological pilot can. Thus, it is possible for the UCAVs to have performance levels, precision levels, situational awareness, and general capabilities that far surpass those of pilots and piloted jets.

This can be mitigated by evolving NN based agents specifically for controlling UCAVs, allowing the NN systems to take full advantage of all the sensory data, and use the UCAV to its full potential with regards to maneuverability. I think that this would give the drone an advantage over standard manned aerial vehicles. To evolve such NN based agents we once again do so through an ALife coevolutionary approach. As discussed in the "Motivations and Applications" chapter, by creating a detailed simulation through a simulator like Gazebo, and creating the simulated UCAVs with high enough detail, and a set of preprogrammed or even evolving fighter jet simulations constrained to the physical limits of the pilot, it is possible to coevolve UCAV controlling NN systems. To accomplish this, we can put two populations of forever warring UCAVs into a simulated 3d environment, to coevolve the ever more intelligent digital minds within. This, as in the Predator Vs. Prey [11] simulations, will yield ever more creative NN based agents, evolving neurocontrollers with innovative combat maneuvers, and having the ability to use the full potential of unmanned combat aircraft, the full potential of metal that is not limited by flesh.

The preliminary testing in this project has started. At the time of this writing, the interface between the DXNN platform and the Player/Gazebo has been developed, and the work is being concentrated on developing simulations of the UCAVs which are modular enough to allow for morphological evolution. Based on the performance of DXNN in ALife, there seems to be no reason why it would not evolve highly adaptive, flexible, and potent UCAV piloting agents.

10.7.3 Evolving New CPU Architectures & Optimizing Existing Ones

The third project currently being pursued by the DXNN research group, deals with the DXNN platform being applied to the evolution and optimization of digital circuits. Because the neurons in the evolving NN topologies can have any type of connections and activation functions, the DXNN platform does not in reality evolve NNs, but Universal Learning Networks, where the nodes can be anything. In this particular application, the nodes use logic operators and transistor simulations as activation functions, thus the evolved topologies are those of digital circuits.

The OpenSPARC project provides the whole architecture and topology of the OpenSPARC T2 CPU, which our team is hoping to take advantage of. The goal of our project is composed of two parts. 1. Create the tuple encoded genotype of a system which recreates the OpenSPARC T2 architecture, and then through its mutation operators (complexifying and pruning), optimize the CPU, by reducing the number of gates used while retaining the functionality. 2. By specifying particular goals through the fitness function, such as increased throughput, higher core count

coherency, and certain new features, evolve the existing architecture into a more advanced one.

Because OpenSPARC T2 also provides a testing suit, it is possible to mutate the existing architecture and immediately test its functionality and performance. But due to the architecture's high level of complexity, the project is still in the process of having new mutation operators being developed, the fitness functions being crafted for optimization and evolution of the CPU, and the creation of the genotype representing the OpenSPARC-T2 architecture. DXNN has been used to evolve and optimize much smaller digital circuits, which gives hope that it can successfully be applied here as well. The potential payoffs could be immense, improving and optimizing CPUs automatically, and adding new features, would revolutionize the landscape of this field. At the moment, we are only beginning to scratch the surface of this project.

10.8 Summary and Future Work

In this chapter we have discussed the DXNN Platform, a general Topology and Weight Evolving Artificial Neural Network system and framework. I briefly explored its various features, its ability to evolve complex NN topologies and its particular approach to the optimization of synaptic weights in the evolved NN topologies. We discussed how DXNN uses the size of the NN in the determination of how long to tune the new synaptic weights, which synaptic weights to tune, and which NNs should be allowed to create offspring and be considered fit. We have also discussed the substrate encoding used by the DXNN, which allows it to very effectively build substrates composed of a very large number of neurodes.

Finally, we have went into some detail discussing the DXNN Research group's current projects. The Neural Network Research Repository, the Cyberwarfare project, the Combat UAV project, and the CPU Evolution project. DXNN is the first neuroevolutionary system built purely through Erlang, and which was designed from the very beginning to be implemented only in Erlang. Without Erlang, something as complex, dynamic, and general as this neuroevolutionary platform, could not be created by a single individual so easily. There is an enormous room for growth and further improvement in this system. And it is this that you and I are working on in this book, we are building the next phase of DXNN.

10.9 References

[1] DXNN's records.hrl is available at: https://github.com/CorticalComputer/DXNN
[2] Sher GI (2010) Discover & eXplore Neural Network (DXNN) Platform, a Modular TWEANN. Available at: http://arxiv.org/abs/1008.2412

[3] Gauci J, Stanley KO (2007) Generating Large-Scale Neural Networks Through Discovering Geometric Regularities. Proceedings of the 9th annual conference on Genetic and evolutionary computation GECCO 07, 997.
[4] Siebel NT, Sommer G (2007) Evolutionary Reinforcement Learning of Artificial Neural Networks. International Journal of Hybrid Intelligent Systems 4, 171-183.
[5] Player/Stage/Gazebo: http://playerstage.sourceforge.net/
[6] Risi S, Stanley KO (2010) Indirectly Encoding Neural Plasticity as a Pattern of Local Rules. Neural Plasticity 6226, 1-11.
[7] Woolley BG, Stanley KO (2010) Evolving a Single Scalable Controller for an Octopus Arm with a Variable Number of Segments. Parallel Problem Solving from Nature PPSN XI, 270-279.
[8] DXNN Research Group: www.DXNNResearch.com
[9] OpenSPARC: http://www.opensparc.net/
[10] DXNN Neural Network Research Repository: www.DXNNResearch.com/NNRR
[11] Prdator Vs. Prey Simulation recording: http://www.youtube.com/watch?v=HzsDZt8EO70&feature=related
[12] Sher GI (2012) Evolving Chart Pattern Sensitive Neural Network Based Forex TradingAgents. Available at: http://http://arxiv.org/abs/1111.5892.