Practical Algorithms for Incremental Growth

Christopher MacLeod
School of Engineering, The Robert Gordon University, Aberdeen

Abstract
This report considers some of the practical issues involved in the implementation of Incremental Growth or Incremental Evolution algorithms as outlined in the paper: *Incremental Growth in Modular Neural Networks* *(doi:10.1016/j.engappai.2008.11.002),* originally published in the journal *Engineering Applications of Artificial Intelligence* and the article: *Minds for Robots*, published in the magazine *Electronics World*. These algorithms allow a Neural Network or similar system to grow, piece by piece, in a controlled manner. The sections below consider the data structures, algorithms and programming techniques which can be used and also addresses unit functionality and possibilities for interesting further work.

1. Introduction

The two papers mentioned in the abstract above[1,2] describe a novel evolutionary algorithm. This algorithm allows the structure of a system to grow piece by piece to any arbitrary level of complexity. In the tests described in the papers, the system used was an intelligent robot. A major facet of the algorithm is that only the piece which is being currently added evolves and so the search space covered in each iteration is kept manageable[3]. A history of the technique can be gleaned by following its development through published papers, from initial ideas[4,5] a PhD project[6] to the first published conference papers[7, 8].

This report was written after several enquiries about the structure of the algorithms and it gives several suggestions for these. However, it should be noted that these are just suggestions and that there are many ways of implementing the system.

2. Network Data Structure

The main Data Structure in an Incremental Growth algorithm is that which contains the details of the system being evolved – in the case of our previous work and this description, a neural network controller. Various different data structures were experimented with and considered for this purpose - for example, linked lists and dynamic objects. However, one structure in particular has several useful advantages:

- It is simple to read, understand and debug.
- It can be directly read by other peripheral programs – for example, routines to display or manipulate the network.
- It can be easily saved and retrieved to a permanent storage device like a hard disc.
- It can be read into working memory or variables for manipulation.
- It can be used with both Object-orientated and Procedural programming systems.
- It allows individual parts of the program to be easily developed and tested in isolation.
- It is particularly easy to use with the growth algorithm.

This structure is the “Netlist.” It is simply a linear list of the neurons in the network. The list may be split up into four parts (or more if necessary):

1. The fixed part of the network (the previously evolved part, which undergoes no further change).
2. The latest module; this is the part of the network which is currently being trained. When training is finished, this gets added to the fixed part.
3. Input list. This holds a list of the inputs and which neurons they’re connected to.
4. Output list. As above, except for outputs.

The “population” of the training GA is made up from a population of 2, 3 and 4 (items 3 and 4 may need a separate fixed and evolvable part as well).

The neural network “engine” simply runs through each neuron in turn and calculates its output.

The structure is shown in figure 1.

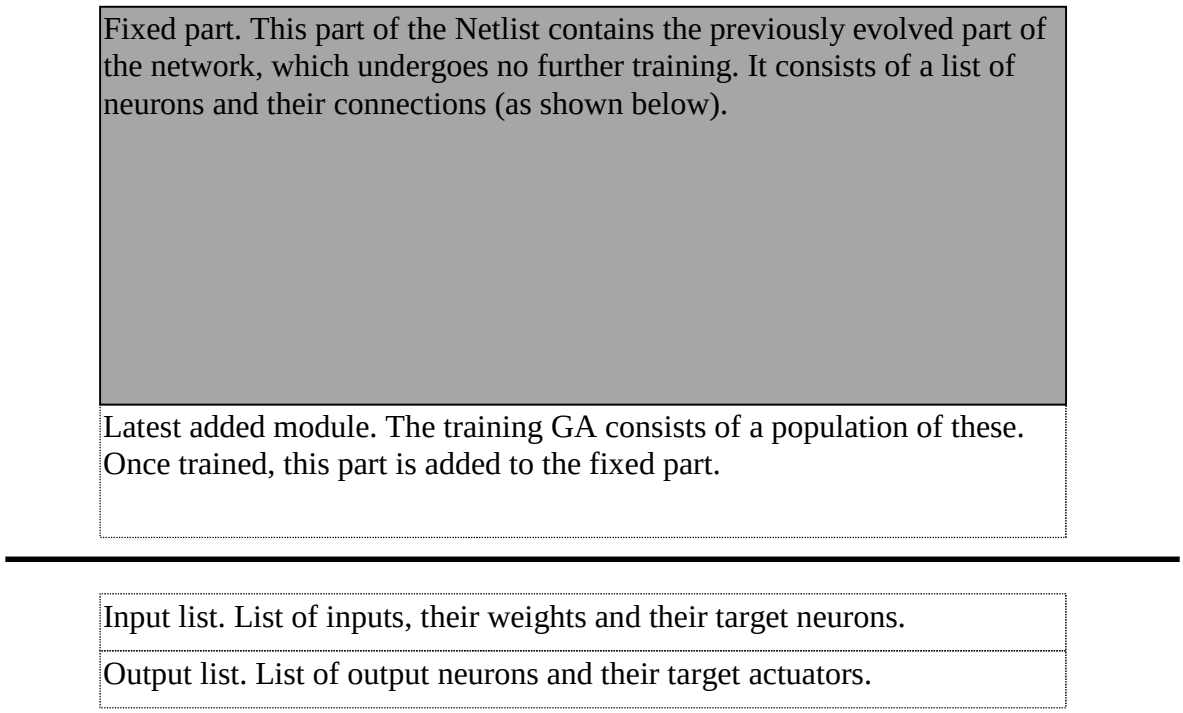

Figure 1, One possible structure for the “Netlist.”

As shown above, the input and output lists might be a separate data structure.

The Netlist is a sequential list of neurons. Showing their parameters and what connects to them. Each entry in the list might be in a format similar to that in figure 2.

Parameter	Neuron number (1)	Neuron Type (2)	Other parameters of neuron (3 to n)	Output of neuron (n+4)
Parameter (continued)	First neural connection (n+5)	Weight (n+6)	Second neural connection (n+7)	Weight (n+8…..x)

Figure 2, Possible structure for a single entry in Netlist.

So the list looks as shown in figure 3.

No	*Type*	*Output*	*Connection1*	*Weigtht1*	*Connection2*	*Weight2……etc*
1	*1*	*0.4*	*6*	*-0.3*	*2*	*0.5……etc*
2	*1*	*0.9*	*3*	*1.8*	*9*	*0.2……etc*
3	*3*	*0.3*	*3*	*3.2*	*4*	*-3.2……etc*
4	*1*	*0.5*	*11*	*0.4*	*3*	*-1.4……etc*
etc						

Figure 3, Appearance of Netlist.

3. Algorithm Overview and Programming

If the program is divided into suitable modules (which can be coded as procedures, functions, methods or separate programs) then each part can be tested and verified independently. One way of decomposing the program in this top-down fashion is shown in figure 4.

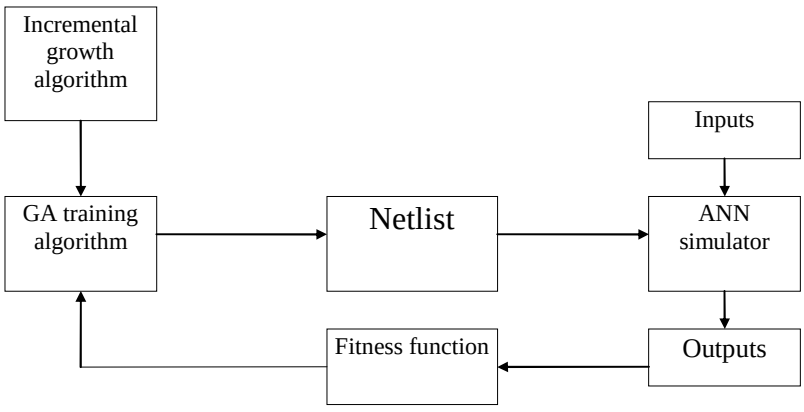

Figure 4, a top-down approach to the program design.

Each of these parts are considered individually below.

A) GA Structure

The purpose of the GA is to define the currently placed module. It must generate both its weights and the connections. In the scheme illustrated above, it would generate a module of the required size (this size parameter is passed from the Incremental Growth Algorithm). The GA then has only to generate the required number of neurons; random parameters; connections and weights and the input / output lists. As

mentioned above, most of the network is fixed and so only a population of the new modules is required. All normal EA operators may apply, although the recombination operator must take into account if different types of neurons have different numbers of non-weight/connection parameters. Because of this, the mutation operator is particularly important and an Evolutionary Strategy type mutation would be recommended. To evaluate the fitness, the GA can concatenate the new module with the previously evolved section and pass control to the ANN simulator section. A pointer can be set to delineate the new module from the rest of the network. A suitable routine is shown in figure 5.

```
Retrieve module size S from Incremental Growth Algorithm.

Generate population of X new modules of size S

Iteration = 0

Do
        N =1
        Iteration = Iteration + 1

        Do
                Concatenate module N with previous (old) Netlist
                Pass control to ANN simulator
                Retrieve, index and store Fitness
                Store Best_Fitness
                Delete module N
                N=N+1

        Until (N==X)

        Select (eg delete worst half of population - if you use roulette, keep best string)

        Recombination (crossover)

        Mutation

        If (Iteration == Max_iteration and Best_fitness < Target_fitness)
        Then report Failure to Incremental Evolution algorithm and pass
        control to IE algorithm

Until (Best_fitness > Target_fitness)

Attach module with Best_fitness to Netlist, shift pointer: Report Success to
Incremental Growth algorithm
```

Figure 5, An implementation of the GA Algorithm.

The algorithm can be configured either to reach a predefined fitness target or report success when the fitness of the current system has plateaued.

B) Incremental Growth Algorithm
This section works hand in hand with the GA. It starts with just one neuron and grows each module up to the point where it can perform the required function as illustrated in figure 6.

It should be borne in mind that, after the system is performing correctly with one particular function, then the whole system, including the robot's body, changes. This aspect of the system has been left out of the algorithm description above for simplicity but is described separately in the next section (in practical systems it would usually be integrated into the Incremental Growth Algorithm (or handled by a separate "over-aching" control algorithm which supervised both controller and body deconstraint).

```
Start:
Size S = 1

Loop:
        Pass control to GA

        If (GA reports Failure)
                Then S = S + 1

        If (GA reports Success)
                Then goto Start (new module pointer could be shifted here if not done
                in GA)

        If (Max_neurons is exceeded)
                Then save Netlist
                report failure to user

Goto Loop
```

Figure 6, An implementation of the IE Algorithm.

C) Neural Nets Engine (simulator)
The Neural Net program would be called from the GA. It would start at the first neuron on the list and run through each one. In the first iteration it would use randomly generated outputs from the other neurons to generate any required inputs which don't have values attached (the same randomly generated numbers each time). As each neuron is run through, its own output is updated, so that from the second generation the numbers converge towards a trained dynamic. The program would also use the Input file to work out its current state. The network would cycle through a fixed number of times (in previous work 500 cycles have been used). The fitness function may be part of this simulator or a different part of the program. A typical Algorithm is shown in figure 7.

```
N=1
Testing_Epoch = 1

Do

        Do

                Check if neuron N is on input list (if it is, add input to activity)

                Calculate activity of neuron N (Connections and weights in neuron's
                list are the inputs to neuron N): Write New_output(N) to list

                Check if neuron N is on output list (if it is, save output to file or
                data structure)

        Until (N = Number_of_neurons)

Until (Testing_epoch = End_of_test)

Pass control to Fitness function
```

Figure 6, An implementation of the ANN simulator.

D) Fitness function
The fitness function is then simply, figure 7.

```
For (Epoch = 1 to End_of_test)

        Read output data file
        Calculate cumulative Error (against previously saved target or fitness formula)

Report Error to GA: Pass control to GA
```

Figure 7, An implementation of the Fitness Function Algorithm.

4. Deconstraint Engineering

The section above deals with the evolution of the neural controller; however, it should be always born in mind that in an Incremental Growth algorithm, the whole system is evolving – the physical layout and environment as well as the controller. In setting this up, the question of the path taken by the system, from simple to complex, is critical. In this section, the physical layout of the system is considered in more detail.

Each individual increase in complexity should be carefully planned and small enough for the training EA to handle. Because the system starts simply and progressively becomes from complex, in introducing further challenges into the environment we are *deconstraining* it from its original simple and constrained form. Therefore, designing an evolutionary path for the system might be termed *Deconstraint Engineering*.

The path of deconstraint which each individual system will take depends on the problem at hand. This may be illustrated by considering the evolution of a prosthetic arm as shown in figure 8.

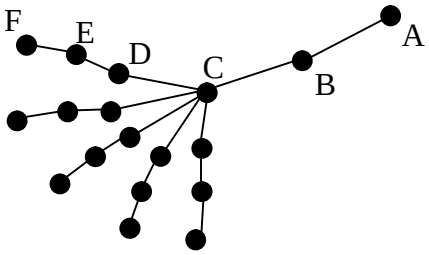

Figure 8, Joint plan of artificial arm

Here, A is the shoulder joint, B is the elbow, C is the wrist, D is the first figure joint, E the second and F the third. The deconstraint path might start with all the joints locked and immovable except B; once B has trained satisfactorily then another joint is unlocked – starting with the simpler joints and moving towards the more complex - like the fingers. Indeed, unlocking one joint at a time like this may not be simple enough, since some joints (for example, the shoulder joint A) are universal ball-types – in this case each universal joint may have to be itself deconstrained, one degree of freedom at a time. We might term a detailed path of deconstraint the *Deconstraint Schedule.*

It should be remembered, that at each stage, three components need to be considered: Firstly, the mechanical form of the system; secondly, its inputs and outputs (sensors and actuators) and thirdly the external environment. At each stage the fitness function of the whole system is also updated to accommodate the current objectives.

So, there are two components to the control-system growth. One is the building up of modules to control a single function and the second is the physical deconstraint of the system. Consider, the example, figure 8 again: It may take several added modules to control the function of B before it performs adequately and the system can be deconstrained by releasing another joint.

5. Simple Experimental Setup

To illustrate the concept of Deconstraint in a practical setting, let us consider an example of a simple experimental setup - in this case using a mobile robot. This was chosen because it is easy to construct and it is informative to make it behave like a simple animal. It would also be a good test-bed to use for developing household robots like vacuuming or lawn-mowing machines.

For practical reasons, the training often has to be done off-line using previously recorded images and a robot simulator. This is mainly because time constraints do not allow us to run the robot with a large population of networks to establish its fitness. However, once the network is trained, it can then be down-loaded into the physical robot for demonstration purposes and there is no reason not to use the robot's own camera to capture the training images – most robotic cameras take images at periodic time intervals anyway.

Consider now an example Environmental Deconstraint Schedule for the robot:

- *Avoid dark shadows* (proceed forward normally but reverse away when in shadow). One of the most basic reactions in simple animals. A simple light / dark sensor is required.
- *Roam and avoid walls* (reverse and turn 180 degrees, or similar behaviour). Robot is placed in "arena" and trained for this behaviour. The sensors required are usually "bump" types, using simple switches.
- *Avoid obstacles* (turn away from object and continue). Obstacles may be painted a different colour to make them more obvious. A simple low-resolution camera (for example a 4 x 4 pixel) can be used.
- *Light seeking* (go towards and sit under light). This is a well-known test for robotic intelligence. Further sensors are unnecessary.
- *Power or food* (identify image object and go towards it). Similar to the above case except using a particular shape. Camera sensor may need to be deconstrained.
- *Danger* (non-moving). As above but fleeing response (an extension of this is a moving dangerous object, this would test if the network can process movement - or mark any moving object as danger).
- *Follow trail* (line following).
- *"Open door"* (task requiring a sequence of movements). In this task we move up a level in complexity. The idea is that when the robot "sees" a particular object it performs a sequence of movements (for example – go backwards, turn, go forwards, turn). This tests it's ability to do complex sequenced tasks (like the metaphorical "opening a door"). Larger networks may be needed to do this.
- *Planning task* (a task involving the development of memory). Again, another step upwards in complexity. The network must find its way (for example) around a fixed maze.
- *Planning task involving dynamic memory* (variable maze, leant with same (fixed) ANN structure). Maze changes and system must learn new path without training. This requires a complex fitness function which rewards the ability to do this.

Notice how the sensors and actuators need to develop along with the environment. This schedule may be conveniently split into four parts:

1. Tasks requiring simple reactions. The first (and perhaps third) task falls into this class because it requires only one action of the robot. All indications are that these tasks should train easily.
2. Tasks requiring (slightly) more complex reactions. The other tasks up to "open door" fall into this category. They require about two actions to be sequenced together (eg reverse and turn 180 degrees). Again no difficulty is anticipated, these tasks serve as a good way of testing and debugging the system.
3. Tasks requiring more than three actions. The "open door" being an example. The purpose of these is to establish the difficulties in evolving such complex sequences. It is possible that this task may provide a "bottle neck" in the work (although it's doubted that it will cause too much problem).
4. Complex reactive tasks - such as the maze - require planning. These are just an extension of the case above and hopefully the previous case will illuminate this

task. However, each module may require a (perhaps initial) large number of neurons to accommodate the sequence and this task may prove a bottle-neck.

5. Finally, the last task. Is it possible to evolve a network which can learn on-line from experience? This task takes the research up to a new level and is likely to be very difficult. In particular, the fitness function will have to be carefully considered to reward this behaviour. Dynamic memory of this sort would put the system on quite a different plane from other ANNs. See section on further work, below.

A typical test robot of the type described above is a DC motor driven, simple chassis with an on-board webcam and PC control.

a) Robot: There is no need for complex electronics in the robot. For example, the tasks above do not require position monitoring (even those requiring planning or dynamic memory). So stepper motors or even dead-reckoning are not needed. Probably a tricycle layout is all that's required. The motors may be switched with relays since speed control is not required.
b) Computer: A PC is the obvious choice. Other than this, a digital I/O board is required for motor interface.
c) Camera: USB Webcams may be accessed from C++ Builder with appropriate DLL files. Code is available for this (some free on the internet). Components are also available for VB, Labview and Visual C++. More technical cameras are also available which allow direct access to their images at the pixel level – for example cmucam. This allows the images used to be initially constrained to a single pixel and follow a deconstraint schedule similar to that outlined in the previous papers. The camera may have to be defocused (made "short-sighted," perhaps using a lens) in order that it doesn't get confused by objects in the distance.

The system is shown in figure 9.

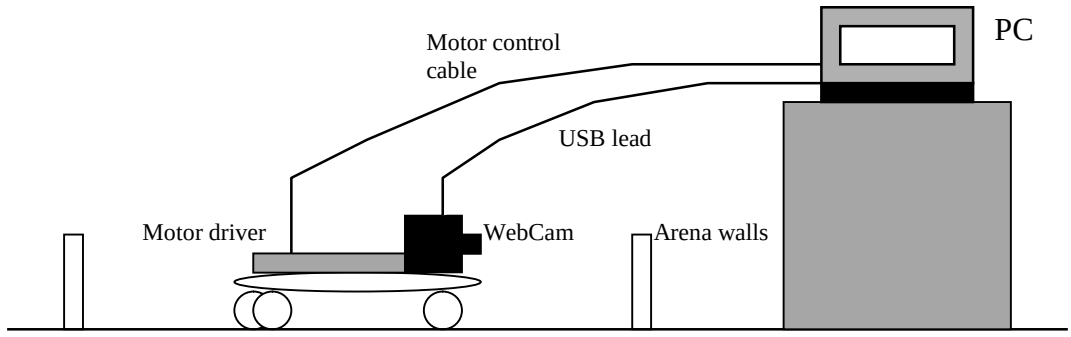

Figure 9, A simple setup for algorithm development and experimentation.

6. Unit Functionality

During the initial stages of designing Incremental Algorithms it was discovered that the functionality of the processing units in the controller was of critical importance. This led to an exploration of unit functionality[12] and so to the adoption of one particular type of unit.

The Artificial Biochemical Network or ABN is the network model which the research group at RGU have used in many of their later experiments. It is well suited to robotic

applications because of two attributes. Firstly, it is easy to program and secondly it is equally at home as a sensory (afferent or input) network, doing tasks like pattern recognition; a control (efferent or output) network feeding actuators like motors or an intermediate (translational, interneuron, hidden or interface) network between these two other types. They also display the sort of time-varying behaviour[13,14,15] which might be important in network dynamics and show that many networked systems – for example, those made of Neurons, Biochemicals and Swarms, can produce parallel systems potentially capable of intelligence.

A summery of the network is given in the 2010 journal paper[9] and earlier publications[10,11] will also provide a foundation. This section addresses the techniques which can be used to programme ABNs.

Although ABNs are conceptually very simple, the main problem in programming them is that, because each unit produces a pulsed output which switches on or off independently in its own time and may be in a different part of its cycle, timing can be difficult. This is easily solved by ensuring that each neuron has its own unique clock variables and the whole network gets executed once in each time cycle. This is illustrated in figure 10.

```
Main loop:
        Neuron 1:
                If neuron 1 is triggered or in operation then
                        Start (or continue with current) action of neuron 1
                        Increment neuron 1 clock
                If neuron 1 is not triggered or cycle is over then
                        Reset neuron 1 clock
        Neuron 2:
                If neuron 2 is triggered or in operation then
                        Start (or continue with current) action of neuron 2
                        Increment neuron 2 clock
                If neuron 1 is not triggered or cycle is over then
                        Reset neuron 2 clock
        .
        .
        .
        .

        Neuron n:
                If neuron n is triggered or in operation then
                        Start (or continue with current) action of neuron n
                        Increment neuron n clock
                If neuron n is not triggered or cycle is over then
                        Reset neuron n clock
End loop
```

Figure 10, the program-structure of an ABN network

In this way each neuron's unique clock gets incremented once per cycle and the individual neuron knows where it is within its own cycle. It is also particularly well suited to the net-list structure outlined in section 2, since this can simply be run through sequentially in each universal time step. Note, however, that the behaviour of the network will strongly depend on the order in which the units are calculated and for repeatable results this must be consistent.

A number of other points are worth making about ABNs, which may make them more useful in some settings. Firstly, if the leaky integrator time constant in the unit is made small, then the unit starts responding to individual pulses rather then the average of many; this is known as "pulse coding" rather then "rate coding" which is used in most units – pulse coding may make the units respond faster and be more sensitive to small changes in timing (which may be a disadvantage as the system might become over-sensitive). Secondly, in the original ABN implementation, the pulse amplitude was constant – however, there may be an advantage in making this vary by either i) including a separate weight for pulse amplitude (solely dependent on unit activation) or ii) allowing the unit to calculate time activation and amplitude activations separately (each with its own weight). Finally, the network can also operate by giving each connection its own delay instead of a weight – the delay allows pulses to co-inside (or miss) with each other, so producing a "delay" encoding; this might have advantages in network dynamics and also in implementing the system in hardware.

7. Suggestions for Further Experiments

A number of interesting applications of Incremental Evolution have been discussed in the literature. This section is a brief overview of these and should provide some ideas for additional experiments and development.

1. Incrementally growing networks onto a pre-existing technological backbone

   One of the potential benefits of an incremental system is that it can be "grown around" a pre-existing system by allowing the grown modules to connect to the previous system's inputs and outputs. The base architecture might be an expert system or a robotic control system such as a subsumption architecture. In the research at RGU, a robotic "backbone" has been developed which is based on biomimetic principles and on the vertebrate Central Nervous System; this is called the Artificial Nervous System[16,17,18,19] or ANS. The system is modular, hieratical and parallel. It is shown in figure 11.

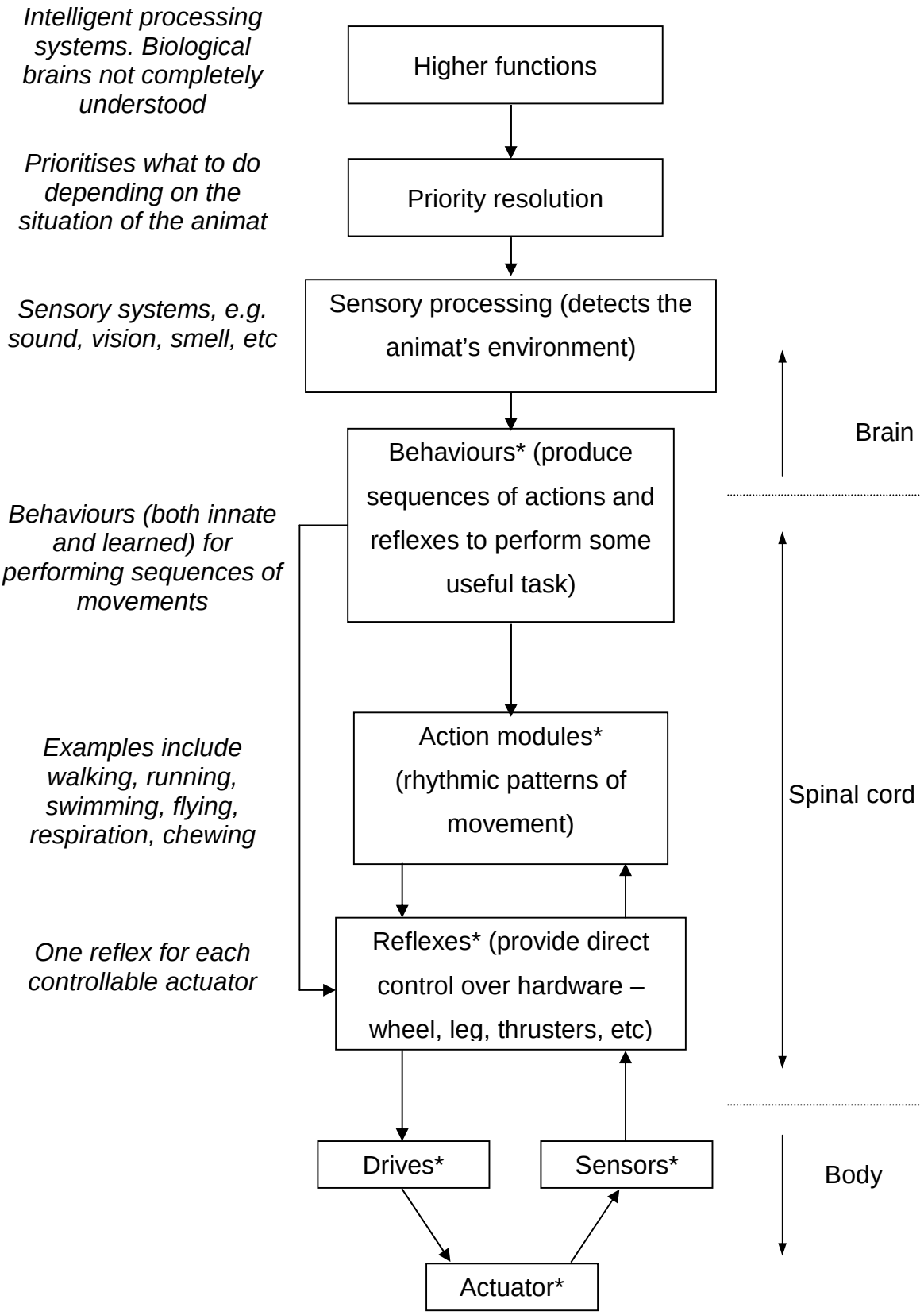

Figure 11, The ANS structure

By allowing the growth algorithm access to any of the inputs or outputs from the system modules, a more complex evolutionary starting point can be obtained. This system is the basis for several on-going projects.

2. Learning and memory in the environment using on-line Incremental Evolution

In its most common form, Incremental Evolution is used to evolve the system's a-priory abilities off-line. However, experiments have been conducted (in a robotic setting) into adding modules as the machine moves around and experiences the world – that is on-line (rather like memory or on-line training). In this scenario, when the robot encounters something worthy of "remembering" a new module is added to its "brain." The main problem encountered in doing this is due to module outputs conflicting – networks respond, not only to their trained pattern, but also to others which fall outside their training set. Generally speaking, this problem can be addressed several ways; these include not connecting the modules in parallel, but placing them in a hieratical structure. Another technique is to train an "allocation" or "voting" module to choose the correct output. Finally, modules may be trained to ignore inappropriate inputs by including these in the training set.

3. Other applications

There are many other applications of Incremental Growth. One example – that of prosthetic control systems, has already been discussed above. Two other areas are worthy of mention. One is in Mechanical Engineering, an example of this is in aerospace design. For example, an aerodynamic control system could be grown starting with a simple shape, other more complex ones being built up on top. The idea is illustrated in figure 12.

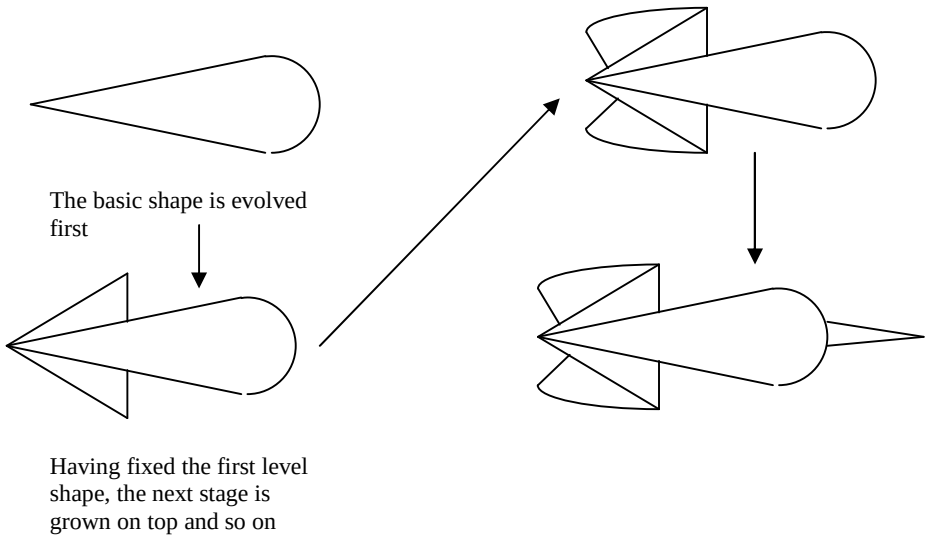

Figure 12, Applications in mechanical engineering – aerodynamics design

The other area is in electronics engineering, particularly in the design of passive networks like filters[2] and matching sections. How incremental growth might be applied to a stub matching section is shown in figure 13.

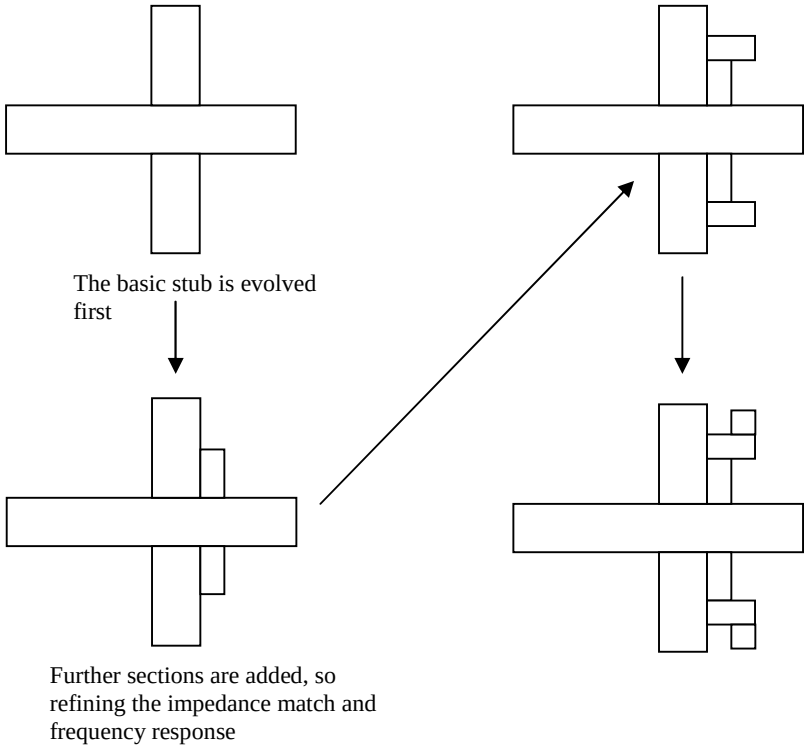

Figure 13, Incremental growth of a broadband microwave tuning stub.

8. Problems yet to be solved

Although the basics of the algorithm are understood and have been outlined in the papers already referenced, there are still some points which need further research – some (although by no-means all) of these are mentioned below:

1. The placement of modules in large systems.

The largest neural network grown in the initial experiments had several hundred neurons in its structure. However, it would be foolish to pretend that one of the possibilities of the method was not to provide a method which could potentially grow networks of thousand or even millions of units. Where to place new modules in a structure like this has not been thoroughly investigated yet. Biology provides some possible solutions to the problem in the layered onion-like structure of the cortex – in this structure newly placed modules maintain contacts only with the most recent prior structures.

Another question related to this is whether (or how) to allow previously placed modules to change. It is likely that as the system gets larger, for efficient development, some changes should be allowed in previous layers. The mechanism for this and its extent is a question which still needs to be addressed. Obviously though, it is important to restrict evolution of the previous system in order to avoid an unmanageable expansion in search space.

2. Placement priorities

In some of the previous experiments, the order in which functions were placed had an important impact on the chances of success. The reasons for this were discussed in the original paper – however, more experiments need to be done before a detailed picture can emerge of the effect.

3. Handling failure and alternative strategies

It has been established in early experiments that the evolutionary direction which development takes depends on the previous deconstraint path and placement of modules (including the issues highlighted above). This can sometimes lead to unsatisfactory progress or failure and may require the system to be "rolled back" or "unravelled" and a new path adopted. A protocol for this has yet to be established.

4. Deconstraint Engineering

Although some general rules for the progress of the system are established, these have yet to be fleshed out and applied to the type of very large systems mentioned above. However, since the path followed needs to be realistic and each stage of development achievable; again, more work needs to be done on this aspect and rules developed which lead to likely paths of success.

Another aspect of deconstraint which has been considered is its automation. In most of the experiments done so far, a decontraint schedule or path has been formulated before-hand - however there is another option. Instead of providing the system with a set body plan etc, at each stage it could choose its own components from an available palette of actuators, sensors, body parts, etc and hence develop in its own unique way without the intervention of the user. Early experiments indicate that this approach is workable. Similarly, at each stage of evolution, the fitness function changes in response to the new task which the system must solve – again it seems possible to have a "universal" fitness function which allows the system to develop in its own direction.

5. On-line Learning

A further area of research is that of on-line learning and memory (as mentioned above). Is it possible to allocate further modules to contain learning or memories as the robot is operating in real-time? The structure of the biological brain has unassigned modules in the cortex which may serve just this purpose. Initial experiments indicate that this is indeed possible – however there are several problems which first need to be overcome. The major one of these is conflict. Modules not assigned to a particular pattern fire when presented with noise and so there is a conflict between two modules – one giving the correct output and another firing for the reason mentioned above. Experiments indicate that there are three approaches to solving this problem: a) To train the new module to recognise noise and ignore it. b) To use a gating module to identify noise or the presence of an important feature and c) To place the modules in a hierarchy in which

information reaches important modules first and is then filtered down towards the less import (this of-course has important implications for the placement of modules in the system).

9. Conclusions

The Incremental Growth algorithm shows great promise in several fields, not the least of which is robotics. In fact, if one considers the issues of search-space and complexity, incremental growth is the *only* obvious system though which very large systems may be built. The basic rules of the method have been established in previous work. The suggestions for further work and discussions above do not imply that the technique has thrown up any unsolvable problems thus far. Instead they merely point to issues which need some thought or work in order to take Incremental growth *to the next level*. Only further work will reveal other issues which haven't been discovered at this point.

References

1. C. MacLeod, G. M. Maxwell and S. Muthuraman, Incremental Growth in Modular Neural Networks, Engineering Applications of Artificial Intelligence, Vol 22, Issue 4/5, 2009. pp 600 – 666, doi:10.1016/j.engappai.2008.11.002.
2. C. MacLeod and G. M. Maxwell, "Minds for Robots," Electronics World, Vol 115, Number 1873, Jan 2009. pp 16 – 19.
3. N. F. Capanni, The Functionality of Spatial and Time-Domain Artificial Neural Models, PhD Thesis, The Robert Gordon University, 2006. pp 23 – 29.
4. C. MacLeod et al, Evolution by devolved action: towards the evolution of systems: In appendix B of D McMinn. Using Evolutionary Artificial Neural Networks to design hierarchical animat nervous systems, PhD Thesis, The Robert Gordon University, Aberdeen, UK. 2002. (also available via on RGU Openair)
5. C. MacLeod and G M Maxwell, "Evolutionary Electronics", Practical Electronics, August 2002, pp578 – 580
6. S. Muthuraman, "The Evolution of Modular Artificial Neural Networks", PhD Thesis, The Robert Gordon University, 2005
7. S. Muthuraman, G. Maxwell and C. MacLeod, The Evolution of Modular Artificial Neural Networks for Legged Robot Control, Artificial Neural Networks and Neural Information Processing, Springer(LNCS 2714), Berlin, ISBN 3-540-40408-2, 2003, p488 – 495
8. S. Muthuraman, C. MacLeod and G. Maxwell, The development of modular evolutionary networks for quadrupedal locomotion, Proceedings of the 7th IASTED International Conference on Artificial Intelligence and Soft Computing, Banff Canada, 2003, p268 - 273.
9. C. MacLeod and N. F. Capanni, Artificial Biochemical Networks: A different Connectionist paradigm, Artificial Intelligence Review, Vol 33, Issue 1/2, 2010. pp 123 – 134, doi:10.1007/s10462-009-9149-y
10. N. F. Capanni, C. MacLeod, G. M. Maxwell and W. Clayton, Artificial Biochemical Networks, CIMCA'2005, Vienna, Austria. Vol 2. p98 – 102.
11. N. F. Capanni, The Functionality of Spatial and Time-Domain Artificial Neural Models, PhD Thesis, The Robert Gordon University, 2006. pp 104 – 201.

12. C. MacLeod, The Synthesis of Artificial Neural Networks using Single String Evolutionary Techniques, PhD Thesis, The Robert Gordon University, April 1999. Appendix 7.
13. K. Hopkin, dot dot dot dash dash dash, New Scientist Magazine, vol 150, 2030., 1996, pp 40 – 43.
14. C. MacLeod and G. Maxwell, Intelligent Signal Processing, Electronics World, December 1999, pp 984 - 987
15. C. MacLeod and G. M. Maxwell, Brains and how to grow them, Electronics World, November 1998, pp 937-940
16. D. McMinn, "Using Evolutionary Artificial Neural Networks to Design Hierarchical Animat Nervous Systems", PhD Thesis, The Robert Gordon University, 2001.
17. C. MacLeod, G. M. Maxwell and D. McMinn, "A Framework for Evolution of an Animat Nervous System", European Advanced Robotics Systems Development: Mobile Robotics, Portugal, September 1998.
18. D. McMinn, C. MacLeod and G. M. Maxwell, "An Evolutionary Artificial Nervous System for Animat Locomotion", Sixth International Conference on Engineering Applications of Neural Networks (EANN 2000), Kingston Upon Thames, July 2000. pp170-176.
19. D. McMinn, C. MacLeod and G. Maxwell, "Evolutionary Artificial Neural Networks for Quadruped Locomotion", International Conference on Neural Networks, (ICANN 2002), Madrid, August 2002, pp 789-794