Impact of Redundant Data on Evolution of Neural Networks

By
Joshua Burroughs

A Thesis
Submitted to the Division of Natural Sciences of New College of Florida, in partial fulfilment of the requirements for the degree of Bachelor of Arts in Mathematics.

Under the Sponsorship of Patrick McDonald

Sarasota, Florida
May 6, 2005

Baccalaureate to be held Friday, May 6th at 3:00 P.M.
in the Math Reading Room.

Contents

List of Figures

List of Tables

Impact of Redundant Data on Evolution of Neural Networks

Joshua Burroughs

New College of Florida, 2005

Abstract

Algorithms were developed and implemented to encode a population of neural networks as "digital genomes". After training, the population is tested, a high performance subpopulation is selected, and individuals of the selected subpopulation are bred. Subject to this routine, neural networks were observed to develop topologies specific to the problem under evaluation. Evolutionary outcomes were studied as a function of the degree of degeneracy in the genetic coding, by statistical analysis of over 4,000 evolutionary experiments.

Patrick McDonald

Division of Natural Sciences

Introduction

What is Evolution?

Evolution is the change, over generations, of the distribution of genes within a population of organisms. It is the reason that an African ape of several million years ago has one descendant who is sleeping in a leaf nest in a tree in Africa, and another who is writing this sentence. One of the primary mechanisms of evolution is natural selection, or "Survival of the fittest"[1]. The phrase, "Survival of the fittest", is the somewhat tautological observation that those organisms which have characteristics most conducive to survival are the most likely to survive (and, crucially, to reproduce themselves).

During the course of its life, any organism will have traits that help it survive and prosper — a longer neck which allows it to reach leaves high in trees; spines on leaves to discourage browsing by giraffes, etc. There are also traits that are detrimental to survival. Any trait which is genetic in nature, and which has a positive impact on the survival chances of an organism which features that trait, will tend to become more prevalent in the population of that species over time.

The mechanism just described is called environmental selection, which is part of natural selection. For many species, including nearly all vertebrates, natural selection has another component: sexual selection. Put simply, unless you are able to produce copies of yourself without involving another party, survival alone does not ensure the propagation of your genes. This means having a mate, and thus being subject to the preferences of the available pool of partners. Over time, traits which are attractive to the opposite sex (or the same, hermaphroditic, sex), tend to become more prevalent in the population.

Evolution can also occur under artificial selection, or selective breeding. Domesticated animals and plants breed (largely) only with human approval, and so the traits that we find most desirable will spread through domesticated populations.

Evolution has produced all the amazing diversity present on the earth today, and in the past. The major flaw, from the perspective of those who would study it, is that it takes a *very* long time. This is one motivation behind the development of evolutionary algorithms.

Evolutionary algorithms are computer simulations, which replicate most of the essential details of evolution as described above. Simulated individuals have genomes which are data within the computer, and which undergo processes which simulate reproduction and mutation. Natural selection must generally give way to artificial selection, but it can resemble the natural variety more closely, by selecting for traits which function in a certain way, rather than specific characteristics (like breeding for an animal which is comfortable in the cold, rather than breeding for a thicker coat).

Two properties of evolutionary algorithms make them especially interesting. They can

[1]This phrase is originally due to Herbert Spencer, but Darwin did use it in "On the Origin of Species"

operate on entities other than biological organisms, such as computer programs, or jet-engine design parameters, making them useful for many types of optimization problems where determining a solution analytically is difficult. They also can proceed at a pace many times faster than natural evolution; even faster than fast reproducing microbes, making them a good way to study properties of evolution that would simply take too long to play out in any biological population.

Neurons and Brain Structure

Now let us turn to the organ of thought - the brain. The brain is composed of billions of neurons, massively interconnected. Each neuron has a single axon which carries its output to the dendrites of other neurons. Each neuron has many such dendrites, receiving input. Computer scientist have used simplified models of neurons, where each neuron takes a weighted sum of its active inputs, and if that sum is greater than a certain threshold, that node's output becomes active. By tuning these weights through a process of training, these 'neural networks' can learn, in a way seemingly similar to our own learning. However, one property in particular of brains (aside from the great complexity difference), is striking in its absence. Usually, neural nets are arranged in simple layers, each usually fully connected (all neurons to all neurons) to the next. However, in the animal brain, different regions have different topologies and widely different functions. For instance, the visual cortex, located at the back of the skull, is used to process visual information. If this portion of the brain is removed, sight becomes impossible, even though the eyes remain in perfect working order. The specific function of many other brain areas is similarly known[2], and although sometimes other portions of the brain can take over a function from a damaged area, the functions seem consistent in every healthy, undamaged brain. This suggests that there is some quality of the arrangement of cells — the topology of the network, if you will — that makes the visual cortex uniquely suited to its job.

Genetic Coding and Redundancy

Nearly every organism has its genes stored using the same code. DNA sequences are composed of four possible nucleotide bases: adenine, guanine, cytosine, and thymine. There are twenty standard amino acids which are used to create proteins, and these must be encoded by DNA. Clearly, representing one amino acid with one nucleotide gives only four possibilities. Using two nucleotides gives 16 possibilities — better, but still not enough. In fact, each amino acid is represented by 3 nucleotides, which makes 64 possible sequences, which is far too many. The solution is that most amino acids are coded for by more than one sequence. This sort of arrangement is called a degenerate code.

Another feature of genetic coding that is of interest is the use of start and end markers, which are sequences of three nucleotides. Every gene is begun by an initiation sequence followed by a start marker, and continues until an end marker. After an end marker, there will be a stretch, before the start of the next gene, which is not part of any gene. This is so-called junk DNA. There are also sections of DNA within a gene, called introns, which also are not translated into protein sequences. Through most of this document, the term intron will be used to refer to any non-coded sequence.

[2]This is largely due to the cataloging of symptoms of patients with localized brain damage

Goals

The original goal of this project was to replicate some part of the features which make, for instance, the visual cortex facile with visual information, and not speech or movement. To accomplish this, neural networks would be subject to evolution. Evolutionary algorithms are sometimes used to set the weights of the interconnections in neural networks, but in this project the connections and neurons themselves were subject to the evolutionary algorithm. The connection weights were set as part of the artificial selection for a particular individual — individuals were selected based on their ability to rapidly learn material presented, and on their ability to then generalize what they learned to previously unseen examples. Trained weights were not passed to offspring.

This project was also influenced by the genetic code of living organisms. A genetic code was developed with features similar to real DNA: a degenerate code, and non-coding sequences. One advantage of, and initial motivation for, this scheme is that procedures to randomly manipulate the genomes of these neural networks could easily be produced, without need for complicated machinations to respect the structure of the encoded information.

Experimental Results

When the first evolutionary experiments were run in this project, some of the early runs were conducted with an added process occurring during the production of each new individual: all redundant data (degenerate codings, and non-coding sequences) was removed from the new individual. These initial experiments tried to evolve networks which were good at learning to count in binary, and when the redundant data was left alone from generation to generation, networks would be produced which were substantially better than their forebears at learning the task presented to them. However, with the redundant data removed at each step, all evolutionary progress came virtually to a halt, with respect to the other mode. This phenomenon was very intriguing, and it became the focus of investigation.

Since the redundant data phenomenon could somehow be tied specifically to the particular problem of binary counting, some other problems[3] were tried. Though not all showed the dramatic difference of the first experiments, the phenomenon seemed to continue to occur. In order to see how widespread it is, and what sorts of problems are most affected, a large collection of problems was needed. What was settled on is the set of all functions, $\{f | f : \mathbb{Z}_4 \mapsto \mathbb{Z}_4\}$.

Road-map

The next two chapters of this document will discuss the technical details, starting with the ways that neural networks were encoded (representations), and the tools used to translate and analyze them in Chapter 1. Chapter 2 documents the details of the evolutionary algorithm used. Chapter 3 discusses the various experiments performed, and presents some analysis of the results. Conclusions and final thoughts follow the end of Chapter 3.

[3]Problem is used to mean a set of inputs, together with the correct outputs for each input.

Chapter 1

Neural Network Representation and Translation

This chapter details the various formats used to represent, compute, and visualize the neural networks used in this project, as well as the tool used to transform a network into an equivalent representation in a different format.

1.1 Network Compilation Infrastructure

The network compiler infrastructure was developed to provide a unified framework for translating between different representations of the same network. The software functions by analyzing the properties of the input network, and transforming it into a standard, internal representation. One of several back ends then produces a file in the desired destination format from this internal representation. Thus, a network can be easily transformed from any recognized input format (there are currently 2), into any output format for which there exists a back-end.

1.1.1 Developmental Motivations

The choice of particular output formats was motivated by several needs. One was to be able to understand how the topology of the networks is changed by the evolutionary algorithm. A back-end to the excellent GraphViz package was created for this purpose, which can show the interconnections, compute order, and feedback status of a network.

Another requirement was networks that could be evaluated and trained quickly, so a back-end was created which could produce c source code specifically for evaluating and training a network, avoiding many table lookups, logic branches, and other overhead that would be involved in a generalized neural network engine.

The final need was for an encoding scheme that could be acted upon by a mutation algorithm with little or no 'knowledge' of how the networks were encoded, and such that the resulting mutants would have a high probability of being viable. The scheme developed is detailed in Section 1.2.2.

1.1.2 Simple Example Network

Through the rest of this chapter, having a single sample network will be convenient for showcasing the different formats for expressing networks. The chosen network is simple to describe: it is a feed forward network, in layers, having two inputs, one output, and one internal layer, fully connected between adjacent layers. Figure 1.1 shows the structure of this network.

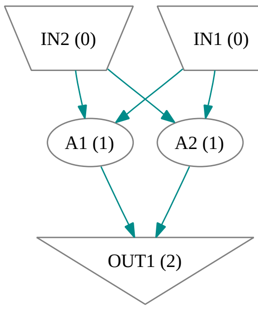

Figure 1.1: A simple network.

1.2 Input Formats

1.2.1 Human Readable Format

A format was needed in order to manually input the configuration of 'seed' networks to be the root ancestors of evolutionary experiments. The syntax will be recognizable to some as Perl data structure declarations; if not, it is summed up in Table 1.1.

The top level of the human readable format is a named value list, with two entries. The first value must be named 'OPTIONS', and contains the fixed options for the network. The second value must be named 'LAYERS', and contains the interconnections and weights, listed in layers.

The 'OPTIONS' value is another named value list, with 'INPUTS' for the number of inputs, and 'OUTPUTS' for the number of outputs. There are currently no other options which can be specified.

The value of 'LAYERS' is also a named value list, where each value is a layer, and the name of the value is the name of the layer. The output layer should be called 'OUT', but other layers can have any name desired. The layers need not appear in any particular order, or in fact be feed forward layers at all.

Each layer value is a list (unnamed values) of nodes. Each node is also an unnamed value list, with each entry being a single connection from another node. Each connection is specified as a two element unnamed value list. The first value is the name of the node from which the input is taken, and the second value is the weight of the connection. 'R' is used to indicate a random starting weight.

Nodes are named sequentially within their layers, by appending the ordinal of the node to the name of the layer (i.e. 'A1', 'A2', 'A3'... in layer 'A'). Below is our sample network in this human readable format:

```
{ OPTIONS => {                          # Start of network options
```

Character	Meaning
{ and }	Enclose lists of named values.
=>	Separates name from value in named value lists. Values can be simple quoted strings, numbers, named value lists, or unnamed value lists.
[and]	Enclose lists of unnamed values. Values are the same as named values.
,	Separates values, or name value pairs in lists.
['LAYERn', WEIGHT]	Defines a single input to a node, from node 'n' in layer 'LAYER', with weight 'WEIGHT', which can be 'R' for random or a real number.
[INPUT, INPUT, ...]	Specification of a single node. The name of the node is the name of the layer, plus a number (the sequence of the node in the list.)
'NAME' => [NODE, ...]	Layer named 'NAME'. Nodes are named NAME1, NAME1, etc.

Table 1.1: Syntax of Human Readable format.

```
            INPUTS => 2,            # number of inputs
            OUTPUTS => 1            # number of outputs
          },                        # end of options
  LAYERS => {                       # start of layer list
           A => [                   # start of 'A' layer
                  [                 # start of node 'A1' (auto #)
                    ["IN1", 'R'],   # IN1 --> A1, random weight
                    ["IN2", 'R']    # IN2 --> A2,   ``     ``
     ],              # end of node 'A1'
                  [                 # start of node 'A2'
                    ["IN1", 'R'],   # IN1 --> A2, random weight
                    ["IN2", 'R']    # IN2 --> A2, random weight
                  ],                # end of node 'A2'
                ],                  # end of 'A' layer
           OUT => [                 # start of output layer('OUT')
                    [               # start of output node 'OUT1'
                      ["A1", 'R'],  # A1 --> OUT1, random weight
                      ["A2", 'R']   # A2 --> OUT1, random weight
                    ],              # end of node 'OUT1'
                  ],                # end of output layer
         }                          # end of layer list
}                                   # end of network
```

1.2.2 Genomic Format

The genomic encoding scheme uses blocks of 32 bits as its basic unit. Each block can be interpreted as a start marker, and end marker, an inter-node connection, or ignored, depending on context. This is the representation operated on by mutation and crossover functions during evolution (see Section 2.1). Figure 1.2 shows our sample network in a genomic representation.

00	00	00	02	2 inputs. (immutable)
00	00	00	02	1 outputs. (immutable)
7c	00	00	00	START node 0
00	01	00	01	random weight from Input 1
00	01	00	00	random weight from Input 0
7e	00	00	00	END
7c	00	00	00	START node 1 (output node)
00	01	00	01	random weight from Input 1
00	01	00	00	random weight from Input 0
7e	00	00	00	END
7c	00	00	00	START node 2 (output node)
00	01	00	02	random weight from Node 0
00	01	00	03	random weight from Node 1
7e	00	00	00	END

Figure 1.2: Simple network in genomic format.

The first two blocks are the number of inputs and outputs, respectively of the network. Since the input patterns and correct outputs have fixed numbers of bits, these numbers are not modified by any mutation or crossover operations.

Start and end markers are any blocks whose 7 most significant bits equal 62 or 63, respectively. This specification was chosen so that these markers would occur with reasonable frequency - 1 time in 128 each. Any blocks after an end marker (or after the immutable options) and before the next start marker are simply ignored, including additional end markers. Nodes are numbered sequentially, by the order they appear, with the input nodes being implicit, and receiving the numbers 0 to $n-1$ where n is the number of inputs. The first node encountered is thus numbered n, the next $n+1$ and so on. The output nodes are always taken to be the last m nodes encountered in the network, where m is the number of outputs.

The blocks after a start marker and before the next end marker are interpreted as connections between nodes. Specifically, each specifies a node from which the current node receives input, and the initial weight of that connection. A connection specifier may also, coincidentally, have a 'start marker' value, but this has no effect within a node.

Each connection specifier has two parts. The 16 least significant bits code for the node from which input is received, with enough high bits being dropped so that the resultant value is less than the total number of nodes in the network (including the implicit input nodes). The 16 most significant bits encode the weight. If either of the last 2 bits of the weight is nonzero, then the connection is initialized with a random starting weight before training. If both are zero, then the remaining 14 bits are treated as a floating point number with a 10 bit mantissa, and a 4 bit exponent.

1.3 Network Analysis Algorithm

This section details the algorithm used to analyze networks prior to compiling them to the destination format. It can detect disconnected networks, nodes which do not affect output, feedback loops, and compute dependencies.

1.3.1 Network Model

The first phase of analysis is to build a model of the network in memory. This model is created as the specialized routines for the particular input network encounter nodes in the input file. Each node is added to the model as it is encountered, and links are created from each node to those nodes it receives input from in a second pass. An important feature of this network model is that links can be traversed backwards.

1.3.2 Tainting

The primary tool of analysis is tainting. Tainting is a system of marking nodes with a 'taint' marker. Each node can be marked with several different taints, but only once per taint. The tainting process begins with a single node, or set of nodes. The starting set is marked, then all links from the starting set are followed, and those nodes are marked. The links are followed from this new set of nodes, and so on. If a node is encountered which already bears the current mark, it is not marked, and its links are not traversed. The process ends when no unmarked nodes are encountered. During a single taint sweep, links are only traversed in one direction. In most cases this is forward, but there is one retrograde sweep.

Taint Sweeps

The first two taint sweeps are input taint and output taint. Input taint starts from the input nodes and proceeds forward. Output taint begins at the output nodes, and proceeds backwards.

After the input and output taints are marked, a taint sweep is performed for each node. For each node in the model, a specific taint for that node only is propagated forward, starting with all the nodes that directly receive output from the target. Thus any node which receives input, directly or indirectly, from the target will have a marker which indicates this fact.

1.3.3 Disconnections

After taint sweeps are performed, the resulting markers can be used to determine characteristics of the network and its nodes. The first of these is disconnection. Any nodes which do not possess both input and output taints is marked as disconnected, meaning that either they do not receive input, or they do not contribute to the output. Disconnected nodes can then be excluded from computation.

Figure 1.3 shows our sample network from Section 1.1.2, with the connection from 'A2' to 'OUT1' removed, creating a disconnect. Input and output nodes are checked for disconnect, which will trigger a warning message, but currently the network is not rejected.

1.3.4 Feedback

It is important to note that when the node-specific taints are calculated, the originating node is not included in the initial taint set. This allows feedback loops to be detected. Any

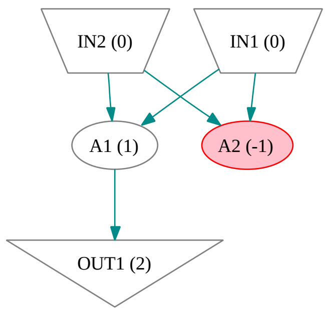

Figure 1.3: Network from Figure 1.1 with one connection deleted. Disconnected nodes are colored pink.

node which bears its own taint is considered a 'feedback' node, since it must (indirectly) receive input from itself. The feedback nodes are also divided into one or more disjoint sets as follows: For each node a which has not been assigned to a partition, the set of all nodes from which a receives (indirect) input is examined. Any of these which is also a feedback node, and which receives input from a is put together with a into a new feedback partition. This is repeated until no unpartitioned feedback nodes remain.

Figure 1.4 shows a slightly more complex network, which features two feedback loops that have been identified. The differing coloration indicates which feedback group the node belongs to. It is easy to see that the network becomes feed-forward if each of the two feedback loops is replaced with a single node.

1.3.5 Determining Compute Order

In order to compute a network, it is necessary to know what order to evaluate the nodes. Taking the set of non-feedback nodes and feedback partitions, which will be collectively called compute units, we can provide an ordering function o, such that $o(a, b) < 0$ if b receives (indirect) input from a, $o(a, b) > 0$ if a receives input b, and $o(a, b) = 0$ otherwise. Since each feedback partition is treated as a single unit, o totally orders the set. The compute units are arranged in the sequence dictated by o, and each unit, starting from the beginning of the list, is assigned a (not necessarily distinct) number. This number starts at zero, and is assigned to each unit in order. It is only incremented when $o(n_1, n_2) \neq 0$. This number is the compute order of the unit.

Figure 1.5 shows a network which illustrates compute orders. Notice how the compute order (shown in parens after the node name) is the same for the feedback loop and the node just to the right of it. Since they do not depend on each other's output, they could be evaluated simultaneously. Also note that the disconnected nodes have a compute order of -1, an indication of their lack of role in the final output.

1.4 Output Formats

This section gives some detail about the formats into which a network can be translated. One of the two major output formats, the genomic format, is also an input format, and has been covered already (Section 1.2.2).

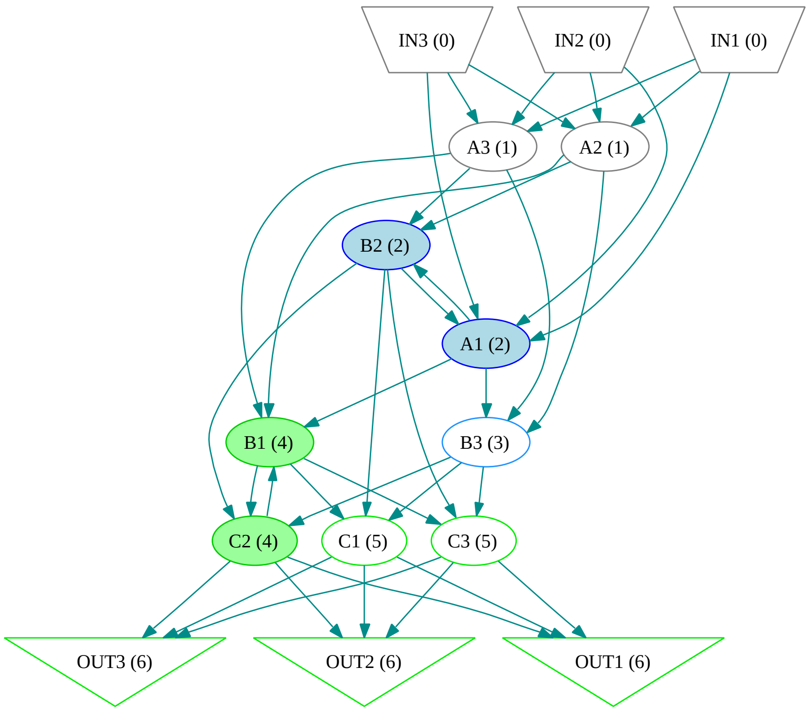

Figure 1.4: Example of a network with feedback. Feedback nodes are filled with blue or green. Nodes with blue or green borders receive input from feedback nodes.

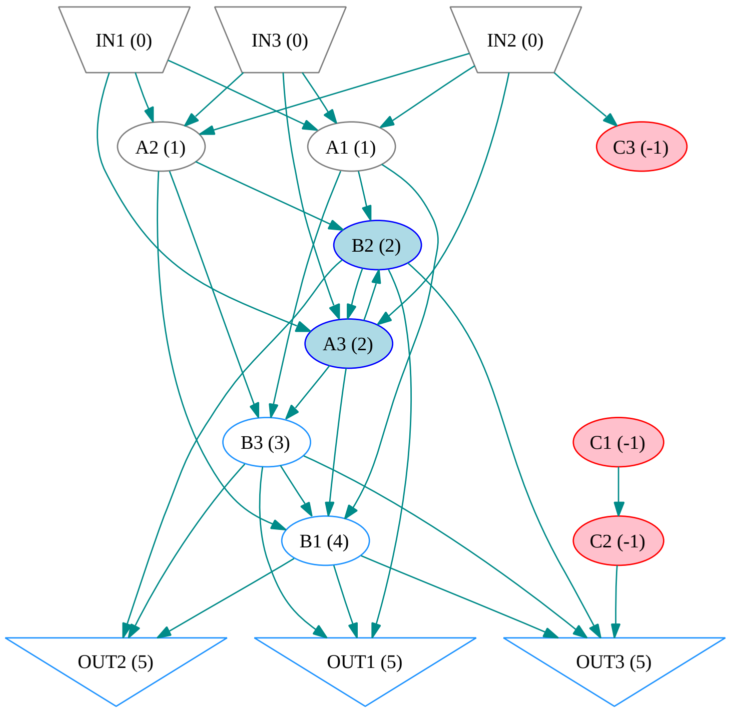

Figure 1.5: Example network, illustrating compute order determinations. Compute order is shown in parentheses after each node's name.

1.4.1 C Source Code Output

The method used in this project to compute and train networks is to compile the network into a C language file, which is used to produce a loadable module for performing computation, etc. with that particular network. The programs produced use standard back-propagation training, with outputs ranging from -1 to $+1$, as given by [3], with modifications for feedback as described later in this section. The network properties discovered through the analysis detailed in the previous section are used during the creation of the C code output, in order to optimize and ensure correct operation. It was hoped that this approach would result in better performance than using an off the shelf package, but that has not been verified.

The simplest use of the analysis is that any disconnected nodes are excised from the network before code is generated. The code also lacks much of the looping which would be required of a more generalized neural network evaluation system. Below is the code for computing the network outputs produced by our simple example net (Section 1.1.2). Other than the incompleteness of this example, the only change is that the formatting has been altered slightly.

```
int
_calc_net (int *inputs, int input_count,
   int *outputs, int output_count,
   double *weights, int weight_count,
   double *node_values, int node_count,
   int feedback_limit, double feedback_convergence)
{
  int i;
  double old_value; //temp. store old value of node to see if feedback has settled
  int feedback_changes; //count of nodes which change over 1 feedback cycle

  //state variables for each node
  double node_IN2;
  double node_IN1;
  double node_A1;
  double node_A2;
  double node_OUT1;
```

⋮

some error checking skipped

⋮

```
  //load input values into input nodes
  node_IN1 = inputs[0];
  node_IN2 = inputs[1];
  ;;;
  /* calculate weighted input for node A1 */
  node_A1 = sigmoid (node_IN2 * weights[0] + node_IN1 * weights[1]);
  /* calculate weighted input for node A2 */
  node_A2 = sigmoid (node_IN2 * weights[2] + node_IN1 * weights[3]);;;
  /* calculate weighted input for node OUT1 */
  node_OUT1 = sigmoid (node_A1 * weights[4] + node_A2 * weights[5]);;;
  //put output values into output buffer:
  //convert from continuous values back to +1/-1
  outputs[0] = ((node_OUT1 > 0) ? 1 : -1);
  if (node_values != NULL)
    {
      //copy all the node values into the array:
      node_values[0] = node_IN2;
      node_values[1] = node_IN1;
      node_values[2] = node_A1;
      node_values[3] = node_A2;
      node_values[4] = node_OUT1;
    }
  return 0;
}
```

Feedback

Feedback loops are treated differently. Before each feedback grouping is calculated, all the inputs for each node in the grouping which does not come from other nodes in the group is pre-summed. Then, for a limited number of iterations, the output of each node in the group is recalculated. During each iteration, the output of each node is compared to its previous output. If the difference for every node in the group is less than a threshold value, the feedback group is considered to have 'settled', and computation proceeds to the remainder of the network. If a group fails to settle before the maximum allowed number of iteration is reached, the values of the outputs after the final iteration are used. Below is the code generated for the feedback group consisting of 'B2' and 'A1' in the network in Figure 1.4.

```
  // BEGIN FEEDBACK GROUP
  //pre-calculate external inputs for all nodes, before feedback loop:
  presum_A1 = 0
    + node_IN3 * weights[6] + node_IN2 * weights[7] + node_IN1 * weights[9];
  presum_B2 = 0 + node_A3 * weights[10] + node_A2 * weights[12];;
  //now main feedback loop
  //init change count:
  for (i = 0; i < feedback_limit; i++)
    {
      feedback_changes = 0;
      old_value = node_A1;
      node_A1 = sigmoid (presum_A1 + node_B2 * weights[8] + 0);
      if (fabs (node_A1 - old_value) > feedback_convergence)
{
  feedback_changes++;
}
      old_value = node_B2;
      node_B2 = sigmoid (presum_B2 + node_A1 * weights[11] + 0);
      if (fabs (node_B2 - old_value) > feedback_convergence)
{
  feedback_changes++;
};
      if (!feedback_changes)
{
  //no node outputs have changed in this loop, so the region has stabilized
  break;
}
    }
  //END FEEDBACK GROUP
```

Feedback is used for training in a similar way. The contribution to error from each of the nodes which receives input from a feedback node is precalculated, then during a limited number of iterations, the error function for each node in the feedback group is allowed to settle, with iteration stopping when all error values change by less than the threshold during a single iteration.

1.4.2 Other Formats

The one other output format currently implemented is GraphViz 'dot' files. Analysis features are more or less directly translated into shape and color specifications in that format. The GraphViz utility 'dot' is used to produce postscript output, which is the source of most of the illustrations of networks in this document. Here is a short snippet of dot-file produced from one network:

```
  B1 [label="\N (4)",color=DodgerBlue];
   A3 -> B1 [color="cyan4"];
   A1 -> B1 [color="cyan4"];
   B3 -> B1 [color="cyan4"];
```

There are some other possible output formats of interest. A network could take on very different properties with the same topology, if the code produced employed a different node model, etc. For greater speed, direct compilation into assembly language, or even machine code is conceivable. Also, the current implementation does not exploit parallelism of the networks. A final possibility is to produce files which could be used to express the network in hardware, possibly using an FPGA (Field Programmable Gate Array).

Chapter 2

Evolutionary Algorithm

An evolutionary algorithm consists of three parts. The first is a population of individuals, each described by some collection of data, known as its genome. For this project, this means the genomic representation of neural networks described in Section 1.2.2. The second is a method for producing new individuals with random variations, by acting on the genomes of existing individuals. The third is a rule for choosing which individuals are selected for reproduction, and which are removed from the population. This generally involves a fitness function, which assigns a numerical score to each individual, based on how closely it embodies the 'target' characteristics of the evolution.

2.1 Network Creation

This section deals with the methods used to create new networks by random operations on existing networks. Two methods were used, mutation and a 'crossover' method to combine the genomes of two networks. In most of the experimental runs, a combination of both was employed, with two parent networks producing a single offspring via crossover, which was then subject to mutation. In biological organisms, the mutations could come before, after, or during meiosis (the process in which crossing over occurs).

2.1.1 Mutation

The mutation method operates on the same 4 byte 'blocks' of which the network genomes are built, and it is aware of the immutable data at the head of each network. Other than these two features, the mutation algorithm does not contain any 'knowledge' of the structure of the data on which it operates. The mutations consist of single block deletions, insertions, and overwrites, and multi block relocate operations.

Relocation choses a random starting point. A number of blocks between 5 and 100 is chosen, and the segment of that length, from the starting point, is relocated to a new point, within about 1000 blocks of its origination.

Insertion adds a single random valued block at any point, and deletion removes a single block. The change operation replaces any block with a new, random value.

Mutations are controlled by a mutation rate parameter, which determines how many mutation operations will be performed per 1000 blocks of genome. For each operation, the choice of type (insertion, deletion, change, or relocation) is uniformly random.

2.1.2 Crossover

The crossover method operates on two 'parent' networks, and is more similar to one-step meiosis than the two-step meiosis prevalent in living organisms [1], which involves an additional duplication of genetic material before the crossing over. The two genomes are, in effect, placed side-by-side. This means that positions within each are equated, based on a scaling so that the beginnings and ends of both networks are matched. The child network is produced by assembling segments taken alternately from each of the parent network. Each segment begins at the location equivalent to the end of the previous segment, and continues for a length between a maximum and minimum value (20 to 750 blocks). This length is derived by choosing a number $x \in (0, 1)$ and the length is then $L_{min} + (L_{max} - L_{min}) * x^n$, where n is a parameter which can be adjusted to favor shorter or longer segments — the value used was generally three, to keep the segment lengths small with respect to the genomes. The resultant network will have a length somewhere between those of its parents.

2.2 Selection

Selection is the process by which individuals are chosen from the population to produce new individuals using the methods described in the preceding section. In order to maintain a constant population, individuals must also be selected for removal from the population to make room for the new individuals.

A simple method, used in some experiments, is to simply select the n individuals with the highest fitness for reproduction. The n individuals with the lowest fitness are then removed from the population. This turnover rate, n, was arbitrarily set to ten (in a population of one hundred) for most of the evolution experiments performed.

A second, stochastic method was employed in most experiments. In this method, individuals were chosen one at a time from the population with a weighted probability. The weighting is equivalent to a box of balls, each bearing the name of one individual, such that the least fit individual is represented by 1 ball and the most fit individual by M balls (usually 10), and all other individuals by a number of balls proportional to the position of its fitness along the continuum between best and worst. This is actually a slight misrepresentation, since the proportions are allowed to take non-integer values. This method was also used for selecting individuals for removal, simply by assigning a relative probability of 1 to the most fit individuals, and M to the least fit individuals, and scaling appropriately.

2.3 Fitness Function

The fitness function is based on performance on a single problem. In any single evolution experiment, the problem (set of inputs paired with correct outputs) is fixed. The problem is divided into a training set and a test set, also fixed.

To evaluate its fitness, a network, with genome specified or random starting weights, is presented each of the inputs in the training set in turn. If it produces the incorrect output for an input, it is trained with the correct outputs. Once a network is successful on all training examples in a single pass, the training phase is complete. There is also a time limit, after which the training period is terminated even if the network has not learned all the examples.

After the training phase, the network is presented with each input in the training set. The correctness of the output is recorded. If the output is incorrect, the fraction of individual outputs which match the correct pattern is recorded.

The amount of time taken for the training process, as a fraction of a second, is also used in the fitness calculation, but since networks generally finish in much less than 1 second, it contributes little to the variability of the fitness values.

The fitness score is calculated as:

$$FIT = 1200P_{train} + \lfloor P_{train} \rfloor (100(1 - P_{second}) + 1200(P_{test} + 0.15(1 - P_{test})P_{partial}))$$

This differs from [5], where the number of iterations required for the network to learn the pattern set was the sole basis for the fitness function.

The networks show a great deal of sensitivity to their random initial weights, so during most experiments, each new network had its fitness averaged over 4 trials, and an additional trial averaged in every generation it survived.

2.4 Genome Manipulation Modes

During evolution experiments, several modes were employed related to the retention of redundant data in the genome during reproduction of individuals. After each new network was created, by crossover of its two parents and subsequent mutation, it could be subject to a procedure to remove redundant data.

The first mode is the 'normal' mode, in which the genome sequences are left unaltered after the reproduction process is complete.

The next mode is 'intron removal', or the removal of non-coding sequences. In this mode, the network genome is scanned block by block, with those blocks which are before the first start marker, after the last end marker, or between a start marking and the subsequent end marker, removed. The data within each node is not altered.

The final mode is 'recompilation'. In this mode, the genome is loaded by the network analysis engine, and a new genome is generated, which represents the same network. Non coding sections are not emitted in this new network, and much other redundant data is discarded as well. For instance, the new version will always use 2 to represent the 2nd node, and never, say, 130 (for networks with fewer than 128 nodes). Start and end markers contain 25 uninterpreted bits, more than 14 bits are unused for any specification of a random initial weight, etc. Nodes can also have blocks which specify duplicate connections, of which all but one copy is discarded. Overall a very large number of bits containing information which is not interpreted is discarded. Table 2.1 shows an example of a network before and after recompilation, to give a visual idea of the amount of information removed.

Original				Recompiled			
00	00	00	03	00	00	00	03
00	00	00	03	00	00	00	03
7c	00	00	00	7c	00	00	00
84	6c	7d	57	00	01	00	03
89	5a	2f	8e	fe	1c	00	05
7e	00	00	00	7e	00	00	00
7c	00	00	00	7c	00	00	00
e2	67	6f	b9	00	01	00	03
05	ab	d9	e9	c7	dc	00	07
1c	a7	b1	b8	7e	00	00	00
7e	00	00	00	7c	00	00	00
7c	00	00	00	00	01	00	08
7e	00	00	00	00	01	00	09
7c	00	00	00	7e	00	00	00
fe	1c	b6	04	7c	00	00	00
e0	17	44	f6	7e	00	00	00
7e	00	00	00	7c	00	00	00
7c	00	00	00	00	01	00	03
ee	cf	d6	76	d0	ac	00	05
f7	cc	8c	b2	00	01	00	04
31	c9	36	b3	00	01	00	02
d0	ac	2d	74	00	01	00	06
02	f6	37	95	7e	00	00	00
25	16	6c	ba	7c	00	00	00
4c	41	20	1b	00	01	00	08
8a	cb	df	c6	00	01	00	05
7e	18	ab	8c	cc	60	00	04
7c	00	00	00	a5	dc	00	07
⋮				⋮			

Table 2.1: Effect of Recompilation on Network Genome.

Chapter 3

Evolution Results

This chapter documents the evolution experiments that were performed, and presents analysis of the results.

3.1 First Experiment - Binary Counting

The first experiment was in evolving networks to perform binary counting. The networks each had seven inputs and 3 outputs. The set of input patterns was simply all 128 possible on/off patterns of the 7 inputs, and the correct output for each pattern was simply the number of 'on' inputs, expressed as a binary number. To make the problem challenging for neural networks, only 3 examples of each number were provided for those numbers with more than one possible input representation (all but 0 and 7). This was never even half the possibilities, and as little as 8% of all possible input patterns for 3 and 4. Table 3.1 shows the number of input patterns and the percentages for each counting number.

Number	Number of Possible Representations	Number of Training Examples	Percent Trained
0	1	1	100%
1	7	3	42%
2	21	3	14%
3	35	3	8%
4	35	3	8%
5	21	3	14%
6	7	3	42%
7	1	1	100%
Overall:	128	20	15%

Table 3.1: Breakdown of binary counting training set, by counting number

An initial population of 100 was created by randomly mutating a single ancestor with 7 inputs, 3 outputs, and one intermediate layer of 7 nodes, and all layers fully connected (Figure 3.1). This network is generally capable of learning the entire training set, but averages only about 35% correct on the test set (averaged over 8 trials with random starting weights), so there is considerable room for improvement.

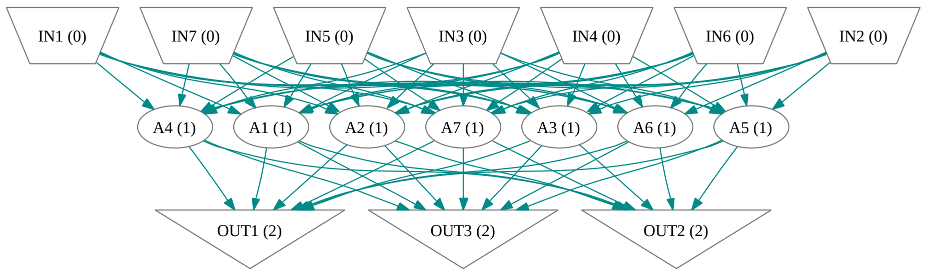

Figure 3.1: Binary Counting Ancestor

3.1.1 Evolutionary Performance

When performing the initial binary counting experiments, networks were recompiled (as defined in Section2.4) after each step. It had not been expected that this would impact evolution, since the recompiled networks were topologically and functionally identical to the originals. However, very poor performance in the evolution was noted. When subsequent changes to this produced a marked difference in the progress of the evolution, it was decided to investigate further by making three configurable modes for evolution as detailed in Section 2.4.

The charts for evolutionary experiments plot several factors versus time (expressed in generations). The light blue lines labeled 'Retained Fitness' show the upper and lower extents of the fitnesses of individuals more than one generation in age. The darker blue labeled 'Average Retained Fitness' is the average fitness of these same individuals. The red line labeled 'Average Overall Fitness' is the average fitness of all individuals in the population at that time. The magenta line labelled 'Average Age' is the average number of generations that the individuals in the current population have been present.

Figure 3.2 shows an example of evolution for binary counting under recompilation mode. Figures 3.3 and 3.4 show evolution of binary counters without genome modifications, and with only introns removed, respectively. These two experiments are much more similar to each other than to the recompilation mode experiment, although the normal mode seems to perform slightly better.

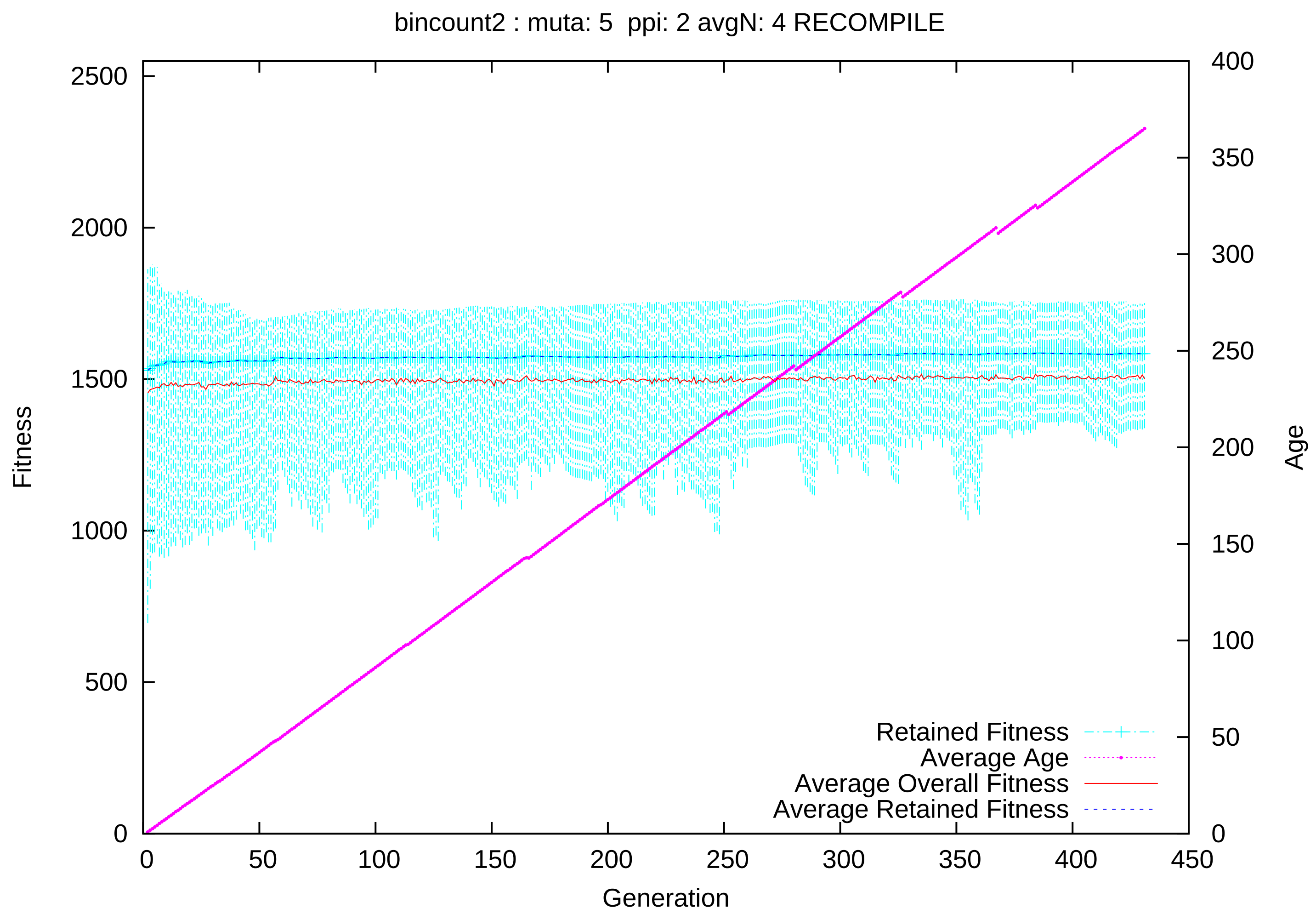

Figure 3.2: Binary Counting Evolution in Recompilation Mode

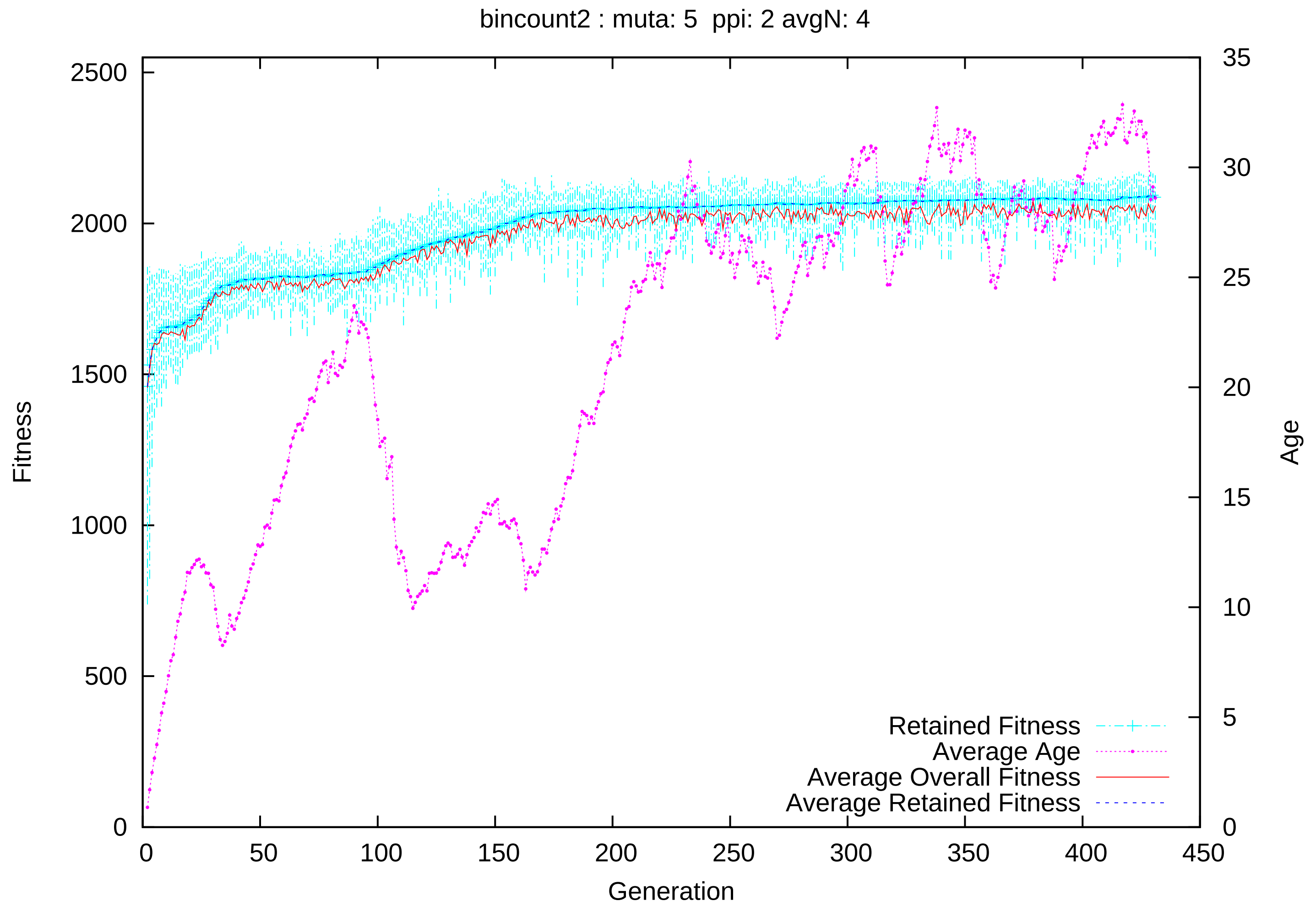

Figure 3.3: Binary Counting Evolution in Normal Mode

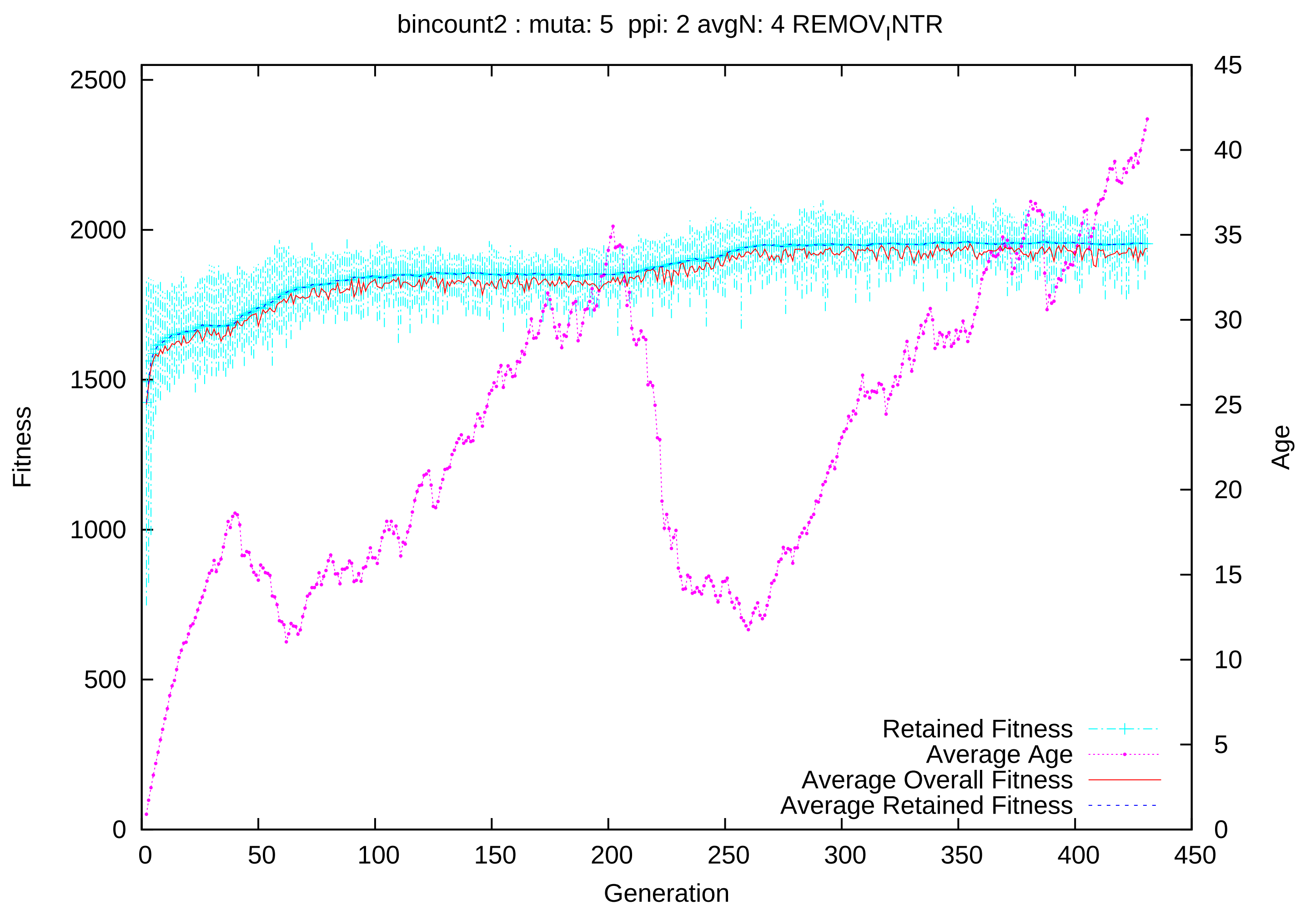

Figure 3.4: Binary Counting Evolution in Intron Removal Mode

3.1.2 Evolved Networks

In subsequent evolution experiments with the binary counting problem, run for up to 50,000 generations, some networks were able to come within 100 points or so of the maximum possible fitness (2500). Figures 3.5 and 3.6 show two such networks. These networks are both descendants of the network in Figure 3.1, but they were produced in completely separate experiments. It is interesting to note that in both cases, input flows forward from the four's place to the two's place to the one's place. In all four long duration experiments, ranging from 13,000 to 50,000 generations, the single networks with the highest fitness in the final populations of each experiment have all shown this pattern.

This arrangement is clearly indicative of the topology of the network becoming specific to the problem for which it has evolved. A plausible explanation for the arrangement is this: The state of the four's place is easy to determine — it must be 'on' if there are 4 or more active inputs, and off otherwise. It does not depend on the state of the other 2 outputs. The two's place, however, must be on if there are 2,3,6, or 7 active inputs, and if the two's place output has access to the state of the four's place output, it can "know" whether to switch on at 2 active inputs, or wait until 6 are active. Similarly, the one's place would use information about the states of both the four's and the two's places, to determine whether to activate after 1,3,5, or 7 active inputs.

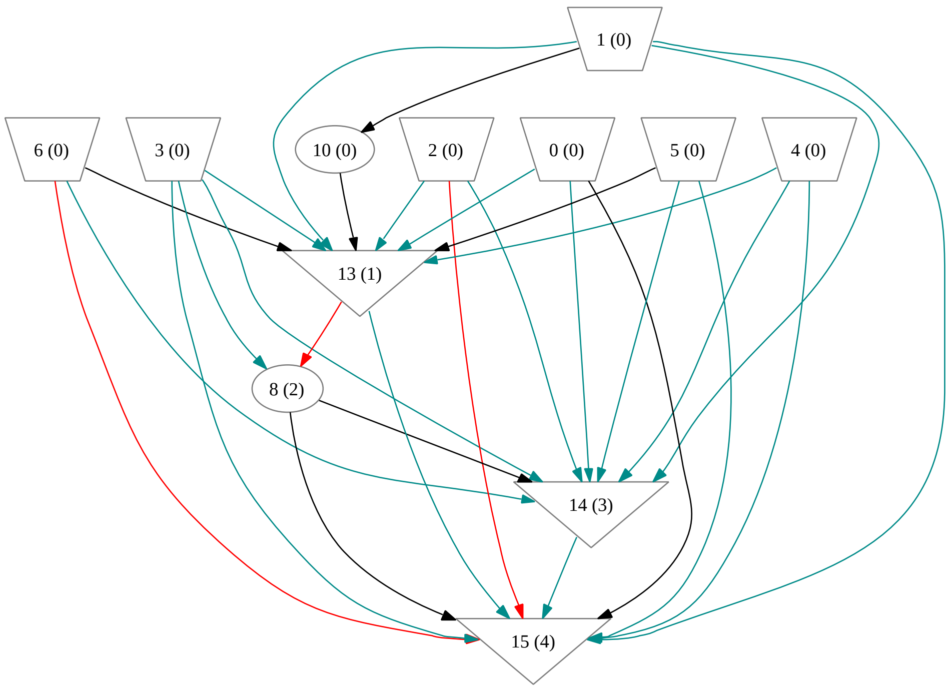

Figure 3.5: Network evolved for binary counting. It earned a fitness score of 2383 out of 2500. Some disconnected nodes removed for clarity. Nodes 13, 14, and 15 are the four's, two's, and one's places, respectively.

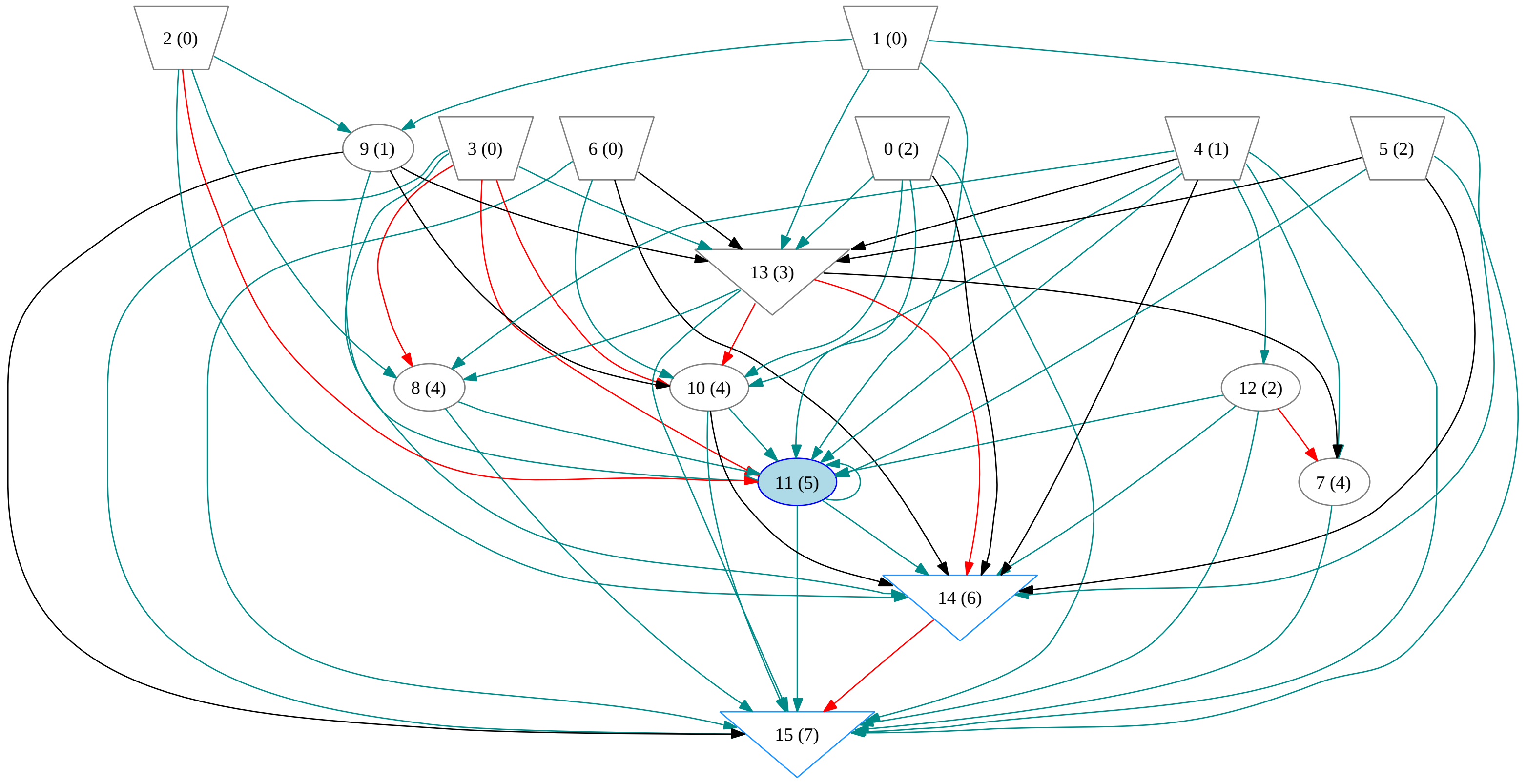

Figure 3.6: Network evolved for binary counting. It earned a fitness score of 2407 out of 2500. Nodes 13, 14, and 15 are the four's, two's, and one's places, respectively.

To attempt to validate the above notions, a network was created by hand to emulate the topology of these binary counting 'winner' networks. It consisted of only the 7 inputs and the three output nodes, with the four's place feeding into the two's and one's place outputs, and the two's feeding into the one's place. This network is shown in Figure 3.7. In six trials, this network averaged 1944.3 fitness, whereas the binary counting ancestor used for evolution experiments averaged only 1307.7, apparently corroborating the importance of the input from higher place values to lower.

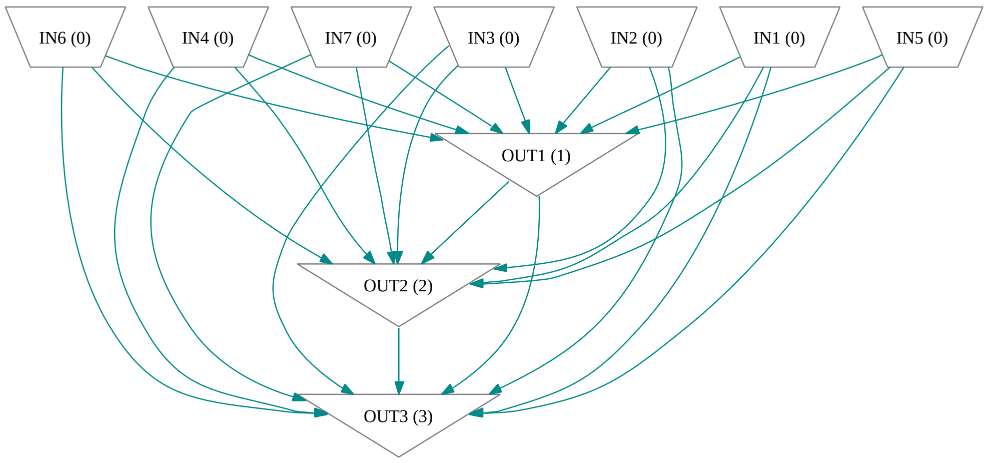

Figure 3.7: Network illustrating successful binary counting topology. This network averaged 1944.3 fitness over 6 trials.

3.2 Examination of Functions $f : \mathbb{Z}_4 \mapsto \mathbb{Z}_4$

It seemed to be the case that removing the random, uninterpreted data from network genomes had a negative impact on the performance of the evolutionary experiments. A set of problems was needed that would allow testing of whether this was a general rule, or something peculiar to the example of binary counting.

The function space of $\mathbb{Z}_4$ onto itself was chosen because it would contain a variety of functions, but have a limited overall size. This was important because each function needs to be evolved once for each genome manipulation mode, and even with optimization, it took 10-20 minutes for one of the personal computers in use to process a single experiment on 200 generations (the number of generations chosen, as it allows time for improvement but is short enough for time constraints on this project). There is also the issue of partitioning the problems into training and test sets. For each problem run in a single mode, two training set partitions were chosen at random - one with a single element training set, and one with a two element training set. Since the function space has 256 elements, a total of 1536 individual 200 generation experiments had to be performed for a single pass. This took over two weeks on a single computer. Three computers were used to make three passes through the set.

3.2.1 Performance Metrics

It would have been impractical to examine the individual progress and performance of the over 4,000 individual evolution experiments run, so some set of analysis metrics was needed to quantify the characteristics of the evolution in each experiment.

It is not entirely clear what value best represents 'performance' of evolution. An obvious choice is the maximum (average) fitness attained ([2] and [4] used population average fitness in the final generation). Another is 'rate' of evolution, as characterized by the slope of the regression line of average fitness versus time. Another possibility is to track significant jumps in fitness over relatively short periods, looking for some sort of 'punctuated equilibrium'. These are the basis for the performance metrics developed to analyze the results of the numerous individual evolution experiments.

All of the metrics were derived from generation-by-generation average fitness values. Three kinds of fitness average were recorded during each experiment:

1. Average fitness of entire population (average fitness).
2. Average fitness of individuals at least one generation in age (retained average fitness).
3. Average fitness of newly created individuals (new average fitness).

The first set of computed values was the minimum, maximum, average, and a normalized average. The normalized average is:

$$\frac{1}{v_{max} - v_{min}} \sum_{i=1}^{N_{generations}} v_i - v_{min}$$

Where v_{min}, v_{max}, v_i are the maximum, minimum, and current value for generation i, respectively. These four quantities were computed for all three averages.

The other derived quantities involved tracking jumps in the fitness averages over intervals of generations. These were computed only for the retained average fitness, and made use of the slope of the regression line of retained fitness vs. generation number. For a 'window' of n generations, the largest single increase between two times n generations apart is found, as well as the slope of this jump (ratio of the jump to the window size), and ratio of the slope of the jump to the slope of the regression line. Also found was the number of non-overlapping n generation windows containing increases in fitness with slopes 2, 3, 5, and 10 times that of the regression line. All these values were computed for 5, 10, 20, and 50 generation windows.

3.2.2 Correlation of Performance Metrics

It is worthwhile to examine the relationships between the various potential measures of performance developed in the previous section. Analysis shows a moderate to strong positive correlation between the maximum attained fitness, the slope of the average fitness regression, and the normed average fitness, suggesting that any of these would be a useful way to measure evolutionary performance.

A large maximum jump is positively correlated to those 3 measures, as well, but counts of jumps which exceed overall slope by a certain multiple are all negatively correlated to the 3 main measurements. Table 3.2 shows the correlation of all values derived from the average 'retained' fitness to the 3 primary values identified in the preceding paragraph.

	Maximum Attained	Normed Average	Regression Slope
Maximum Attained	1.000	0.165	0.414
Average	0.843	-0.211	-0.088
Normed Average	0.165	1.000	0.817
Regression Slope	0.414	0.817	1.000
5 Generation			
Max Jump	0.347	0.782	0.703
Slope	0.347	0.782	0.703
vs. Regression	-0.046	-0.084	-0.112
Count			
2x Regression	-0.035	-0.117	-0.147
3x	-0.176	-0.239	-0.369
5x	-0.260	-0.472	-0.622
10x	-0.263	-0.477	-0.620
10 Generation			
Max Jump	0.319	0.859	0.739
Slope	0.317	0.863	0.738
vs. Regression	-0.049	-0.081	-0.112
Count			
2x Regression	-0.422	-0.662	-0.891
3x	-0.402	-0.684	-0.919
5x	-0.345	-0.558	-0.764
10x	-0.246	-0.428	-0.573
20 Generation			
Max Jump	0.288	0.942	0.756
Slope	0.288	0.942	0.757
vs. Regression	-0.060	-0.084	-0.124
Count			
2x Regression	-0.405	-0.718	-0.916
3x	-0.354	-0.584	-0.788
5x	-0.283	-0.435	-0.629
10x	-0.192	-0.340	-0.464
50 Generation			
Max Jump	0.315	0.966	0.840
Slope	0.315	0.966	0.840
vs. Regression	-0.061	-0.075	-0.114
Count			
2x Regression	-0.349	-0.581	-0.752
3x	-0.286	-0.355	-0.590
5x	-0.209	-0.289	-0.477
10x	-0.125	-0.221	-0.314

Table 3.2: Correlation between selected potential performance metrics over 4353 evolution experiments. All values are derivates of the per-generation average fitness of individuals greater than 0 generations in age.

3.2.3 Complexity

Initial examination of the experimental results indicated that some problems were more conducive to evolutionary progress than others, and so some measure of the difficulty of each problem seemed needed. A simple notion of boolean complexity was used, applied to each output singly. For the boolean function represented by a single output, the number of bits of the input which needed to be examined was used as the complexity value. The single output could require 0 (for constant output), 1, 2, or $\frac{3}{2}$. If an output requires $\frac{3}{2}$ inputs, this means that for half of the possible input patterns examination of a single input suffices, and both must be examined the other half of the time. Table 3.3 shows the possible patterns for a single output, and the number of inputs which must be examined to determine the correct output.

	Inputs A,B					Number
Pattern	00	01	10	11	Description	Bits
0	0	0	0	0	always 0	0
1	1	0	0	0	NOR	$\frac{3}{2}$
2	0	1	0	0	$\bar{A} \wedge B$	$\frac{3}{2}$
3	1	1	0	0	$\bar{A}$	1
4	0	0	1	0	$A \wedge \bar{B}$	$\frac{3}{2}$
5	1	0	1	0	$\bar{B}$	1
6	0	1	1	0	XOR	2
7	1	1	1	0	$NAND$	$\frac{3}{2}$
8	0	0	0	1	AND	$\frac{3}{2}$
9	1	0	0	1	$\overline{XOR}$	2
10	0	1	0	1	B	1
11	1	1	0	1	$\bar{A} \vee B$	$\frac{3}{2}$
12	0	0	1	1	A	1
13	1	0	1	1	$A \vee \bar{B}$	$\frac{3}{2}$
14	0	1	1	1	OR	$\frac{3}{2}$
15	1	1	1	1	always 1	0

Table 3.3: List of all 2 bit boolean functions, showing number of input bits examined to produce output. $\bar{X}$ denotes the negation of X.

To compute a complexity score for each function in $\mathbb{Z}_4 \mapsto \mathbb{Z}_4$, the two individual scores for each output were simply added.

3.2.4 Performance vs Complexity

To examine the effects of complexity and compilation mode on evolutionary performance, we compare how performance varies with complexity in each of the three evolution modes. Figures 3.8 and 3.9 show the slope of the average fitness regression versus complexity, and 3.10 and 3.11 show maximum attained average fitness versus complexity. In 3 of the four charts, the recompilation experiments show a stronger negative correlation to complexity.

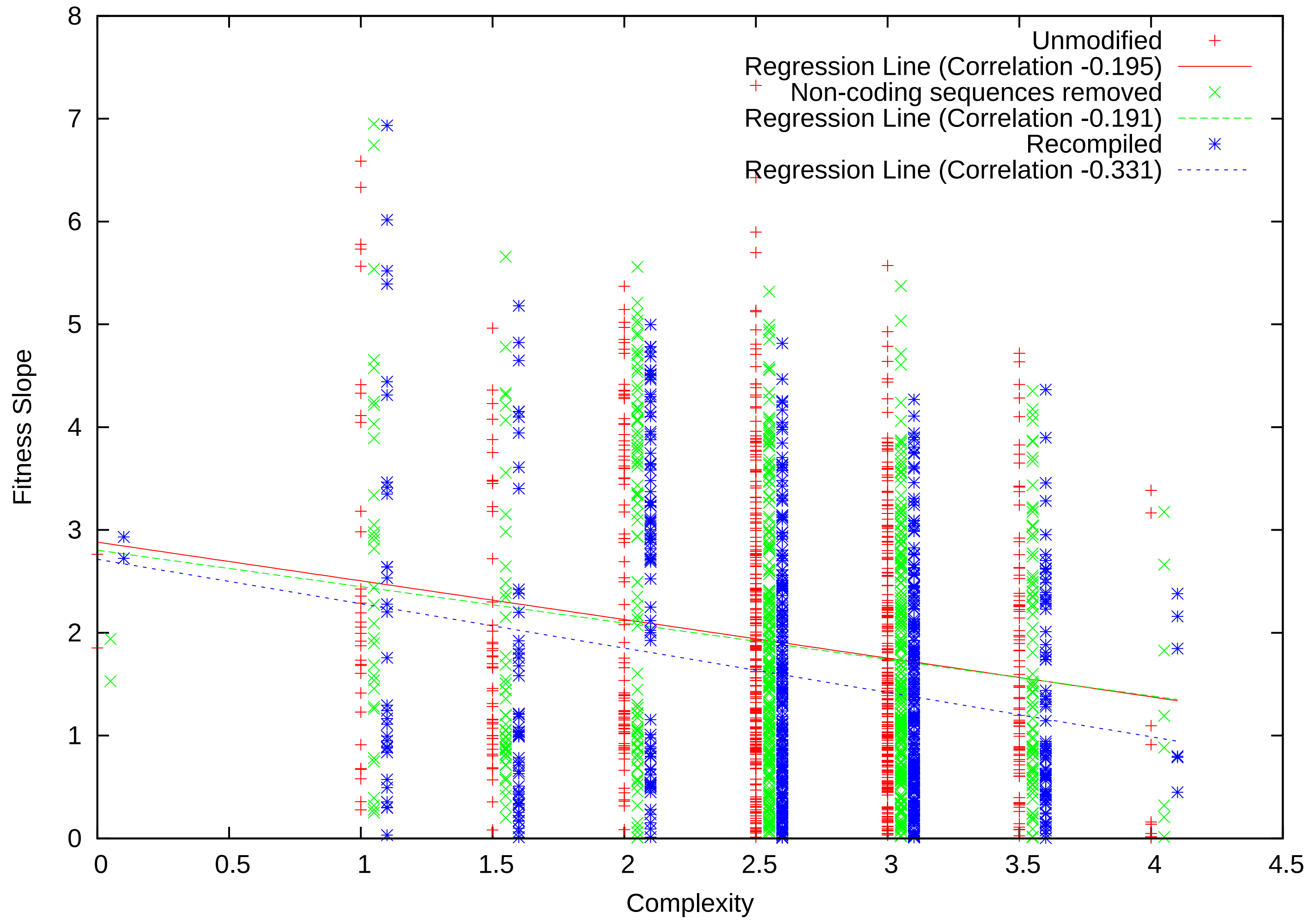

Figure 3.8: Complexity versus Evolution Rate in 3 Modes, with one-pattern training sets. Evolution rate is the regression slope of average retained fitness.

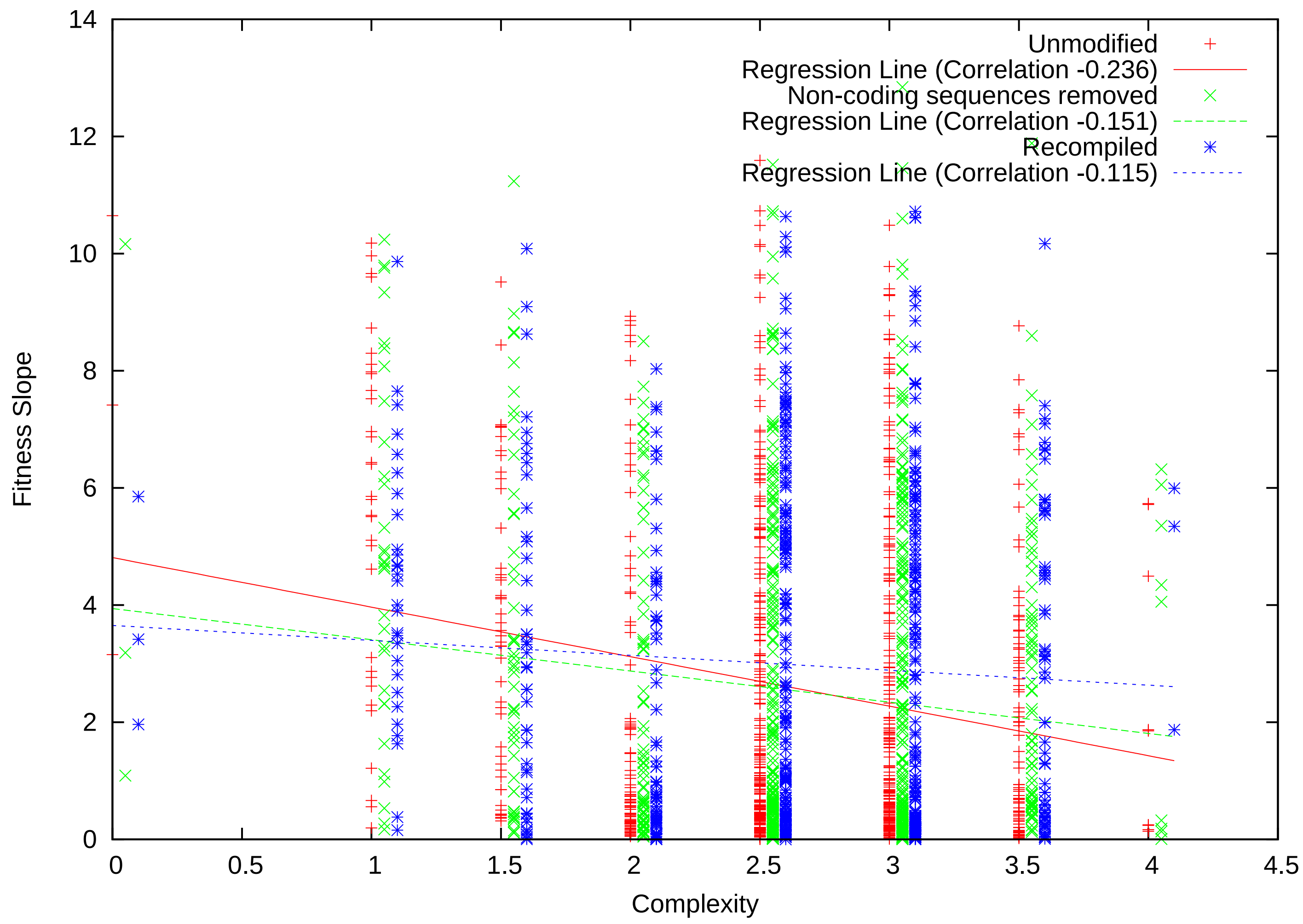

Figure 3.9: Complexity versus Evolution Rate in 3 Modes, with two-pattern training sets. Evolution rate is the regression slope of average retained fitness.

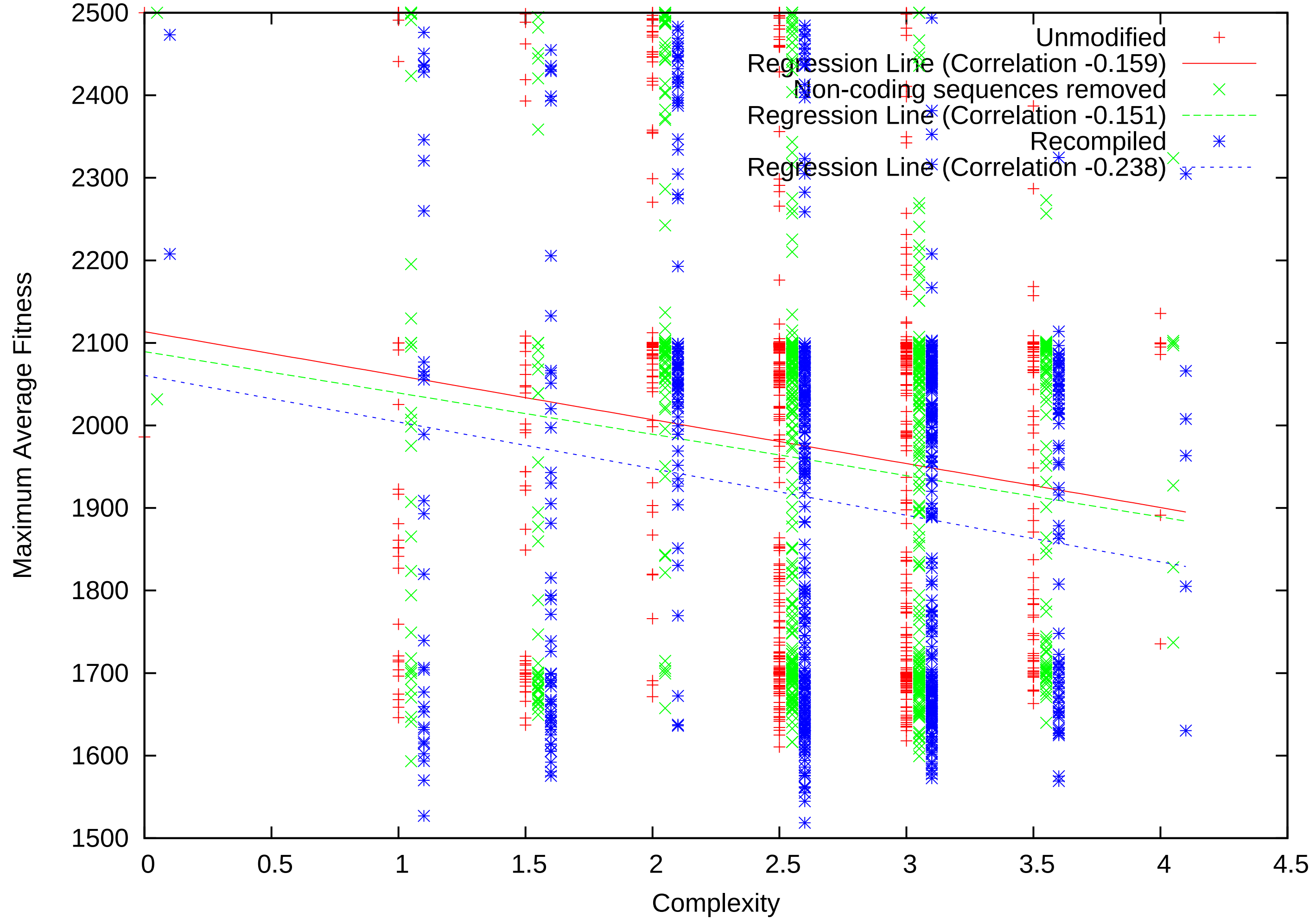

Figure 3.10: Complexity versus maximum attained fitness in 3 Modes, with one-pattern training sets.

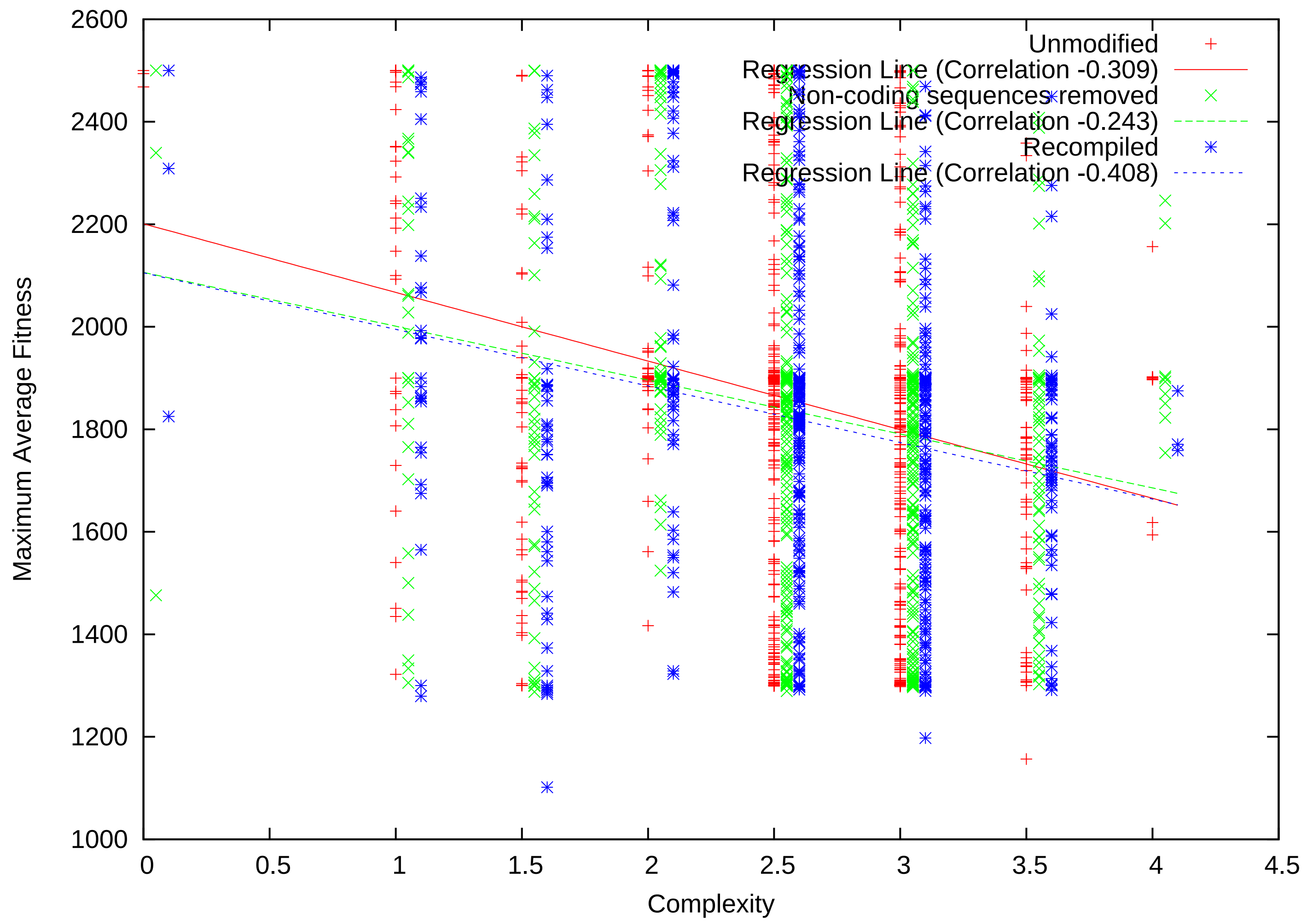

Figure 3.11: Complexity versus maximum attained fitness in 3 Modes, with two-pattern training sets.

Conclusions

In the experiments performed in this project, the topology of the neural networks appeared to have a strong influence on their ability to learn and generalize on certain problems. Topologies were developed which were highly specific to a particular problem, that of binary counting. A surprising degree of convergence was observed over the four long-duration (13,000 to 50,000 generation) experiments in binary counting networks, with all of the highest fitness individuals from each experiment sharing one particular topological characteristic related to the values of the output bits. These results correspond, as suspected, with the highly specific and specialized functionality of disparate regions of the brain.

Also, in the experiments performed, higher complexity (as calculated) had a generally adverse effect on the progress of evolution. The hypothesized relationship between the removal of random/redundant data in the genomes and the performance of evolution was also borne out in the experiments.

The strongest effect of the evolutionary modes was seen when the networks were given only a single input pattern out of 4 on which to train. These experiments were more similar to the binary counting experiments, in which networks only trained on 15% of the total problem space.

It is the opinion of the author that the accumulation of random changes in parts of the genome which do not directly affect the expressed 'phenotype' forms a sort of potential. At any time, a mutation could cause some of the unexpressed data to be switched on, so to speak, and become active parts of the phenotype. In these pieces of data carried from generation to generation, a set of changes can be accumulated which will occasionally be 'tested' by conversion to expressed genetic data.

Of course, this 'testing' will usually produce offspring which are non-viable, or of reduced fitness, but the unexpressed data can continue to develop in other individuals, until it develops a form which allows it to become a beneficial mutation when activated.

Further Study

Although it appears that non-coding data in some way assists the evolution of these neural networks, the exact mechanism is unclear. It is also unknown whether this principal is more generally applicable, what kinds and how much non-coding data are optimal, and whether different levels of degeneracy are better for different evolution systems. A system which allowed for other kinds of artificial genetic codes to have variable amounts and kinds of redundant data would be useful for investigating these problems. The new coding scheme(s) could allow for varying levels of degeneracy to be inherent to the actual encoding, rather than being produced by active modification of the genetic code.

In particular, if we think of the encoding as a map from the n encoded possibilities (call this set E) to the m possible codons (G), in addition to the relative sizes of E and G, it would be interesting to control for other qualities. For instance, is the image of $x \in E$ a set of 'adjacent' values (adjacency should probably be related to the mutation operation — i.e. mutational distance), or if not, how widely scattered are the elements of the image? Furthermore, if the encoding is a map from vectors in E to vectors in G, the 'shape' of the mapping could vary in all sorts of ways, such as allowing encoded pieces to overlap.

Of course, for any scheme for translating a genetic code into an expressed individual/fitness value, some kind of measure of difficulty might be useful, though examination of the differences between the entire groups of different degeneracies might still show a significant difference.

The other aspect of this work is the topological evolution itself. Tools to analyze the ways in which the evolved neural networks solve the problems they are evolved for could be developed. For instance, the ability to inspect the complete internal state of a network when presented with each input pattern could provide substantial insight.

Code could be developed to determine the degree of similarity between two networks. Taking a single problem as the subject of a number of long (20,000 generations or more) evolution experiments, the final populations of all experiments could be combined, and the fitness distribution computed. The high-fitness outliers on this distribution could be examined for similarity. One would expect to see one or more tight clusters, representing the possible 'good' strategies.

Another interesting project would be to evolve networks for two or more problem sets simultaneously. Perhaps a better 'generalized' network could be evolved.

Different types of neural networks could be evolved, as well. The evolution tools created in this project simply mutate and breed directed, weighted graphs; a different method of evaluating fitness, by interpreting them as something other than a neural network could be substituted.

Appendix A

Supplemental CD-ROM

Accompanying this thesis is a supplemental CD-ROM containing the source code created for the project, and data files and neural networks generated during the various experiments discussed in this document. This Appendix provides a listing of the files and directories included, with comments. This documentation generally assumes you are using these files on some sort of Unix like operating system, that you are using ksh or bash, and that you are familiar with its operation, and with the Unix command line environment. You will also need to have a few other utilities installed on your system to make full use of the files, such as gnuplot, bzip2, pstoimg, and GNU GhostScript for the gv utility.

File/Directory Name	Description
/	The root directory
Makefile	Recursively calls 'make' on every subdir with a Makefile
setup_environment	Sets up environment variables need to build and run the programs in this package. Must be 'sourced' into your shell, by typing "`. ./setup_environment`" from this directory. Will not work on csh.
/bin **/include** **/lib**	These directories are empty, but executables, libraries and header files will be placed here during a 'make' of the complete project.
/docs	Contains printable copies of this document
thesis.pdf	PDF format; use Adobe Acrobat Reader.
thesis.ps	* Postscript format; view with ghostview, or send directly to postscript compatible printer.
/networks	Human editable example networks and compilation tool
Makefile	Rules for 'make' to build .so files for all networks in this directory
compile.pl	Simple tool for producing c source from a .net file. Uses /perllib/NetCompiler.pm
binary_counter.net	Basic 7 input, 3 output, one hidden layer network, used as the ancestor for binary counting evolution.
binary_counter_ng.net	Another net for binary counting, created using ideas gleaned from descendants of the previous file.
	continued on next page

continued from previous page

File/Directory Name	Description
test0.net test1.net test_2by2.net test2.net test3.net test4.net test5.net test6.net testfb.net test.net	Other test/example networks
/neural	C source code for libneural.so (Appendix C)
Makefile	Rules for building libneural.so with 'make'
error.c	Error handling routines
neural.c	Main interface
sets.c	functions for batch operations on input/output pattern sets
/neural/include	Header files for libneural
neural_err.h	Prototypes for internal error handling functions. Client code does not need to include this header.
neural.h	Main header for libneural
/perllib	Perl modules (Appendix B)
Errorable.pm	Error handling base class module
Evolver.pm	Genetic Algorithm implementation
Mutation.pm	Performs mutation and crossover on binary network genome files.
NetCompiler.pm	Package for translating neural networks between different file formats.
NetEvolvee.pm	Wraps neural networks for use with Evolver.pm
ProjectConfig.pm	Caching configuration file wrapper
Utility.pm	Utility function catch-all
/perllib/NetCompiler	Output modules used by NetCompiler.pm
C.pm	Produces C source code which 'runs' a neural network
Genome.pm	Produces binary network genomes
GraphViz.pm	Produces GraphViz '.dot' file pictures of networks
network.c.tmpl	Template Toolkit template used by C.pm
/perllib/NetEvolvee	Support files for NetEvolvee.pm
evolve_makefile	'make' rules used to compile networks
/projects	Configuration files, networks, data, and analysis tools from evolution experiments
analyze_logs	Produces statistics from a set of Evolver.pm data log files; one set of numbers per log file.
cleanup	Cleans out files from an Evolver.pm project dir and does some initialization.

continued on next page

continued from previous page

File/Directory Name	Description
complexity_plot	produces a gnuplot graph of a given field versus the complexity score from an analysis file output by analyze_logs.
compute_log_avg_stddev	computes average and standard deviation of quantities listed in analyze_logs output.
datalog_dump.pl	Extracts fields from an experiment data log.
datalog_reader.pl	Dumps data log file to screen
do_analysis	Runs analyze_logs on the logfiles from bigproj_data.tar.bz2
do_metanal	Runs compute_log_avg_stddev on output from do_analysis
plot_all	calls plot_project.pl on a list of files
plot_project.pl	generates a gnuplot chart of evolutionary progress from an experiment log
pngplot	turns a set of .ps graphs into .png files
regress_plot.pl	Does a scatter-plot of two values in a file output by analyze_logs, and also draws the regression line
regress_tbl.pl	Creates a table of correlation coefficients between values in analyze_logs output
runbigproj.pl	Runs batches of experiments from the $\mathbb{Z}_4 \mapsto \mathbb{Z}_4$ problem space
run_project.pl	Runs an experiment in a give project dir for a specified number of generations and repetitions
bigproj_data.tar.bz2	Data from all $\mathbb{Z}_4 \mapsto \mathbb{Z}_4$ experiments performed. Unpacks to 2.4GB of data
bigproj.tar.bz2	Project dir for $\mathbb{Z}_4 \mapsto \mathbb{Z}_4$ experiments
bincount2_data.tar.bz2	Data from 2nd set of binary counting experiments
bincount2.tar.bz2	Project dir for 2nd set of binary counting experiments
bincount3_data.tar.bz2	Data from 3rd binary count experiment
bincount3_long_runs.tar.bz2	Data and networks from long duration (20,000+ generation) binary counting experiments
bincount3.tar.bz2	Project dir for 3rd binary counting experiment
bincount.tar.bz2	Project dir from 1st binary count experiment
/projects/bigproj_metadata	Contains data from analysis of logs in bigproj_data.tar.bz2
bigproj_a1_runs	generated by /projects/do_analysis
bigproj_a2_runs	
bigproj_allruns	
bigproj_a_runs	
bigproj_b1_runs	
bigproj_b2_runs	
bigproj_b_runs	
bigproj_c1_runs	
bigproj_c2_runs	

continued on next page

continued from previous page

File/Directory Name	Description
bigproj_c_runs	
bigproj_a1_meta bigproj_a2_meta bigproj_allmeta bigproj_a_meta bigproj_b1_meta bigproj_b2_meta bigproj_b_meta bigproj_c1_meta bigproj_c2_meta bigproj_c_meta	generated by /projects/do_metanal
/src	Source code for some command line utilities
Makefile	
make_train_and_test_set.pl	translates Perl data structures into a format suitable for `fread_net_io_set()`. See section C.2 for more on this function, and the directory /training_files for sample input files.
train_and_evaluate.c	Trains a network .so, and tests it, using the sets found in a file like those output by make_train_and_test_set.pl. Prints statistics about the training and testing.
/test	Various simple test scripts
ctest.pl	test C compilation
genometest1.pl	checks that genome format contains equivalent network
genome_wght.pl	tests encode and decode of 16 bit weight specifiers in NetCompiler::Genome
make_example_mutations	
/tools	Various command line tools
Makefile	Called by root dir Makefile to copy files to /bin
imggen.pl	Processes a network through NetCompiler::GraphViz to produce a picture
mk22_all.pl	Produces training files for $\mathbb{Z}_4 \mapsto \mathbb{Z}_4$ problem space
mkbincount.pl	Used to create the binary counting training file
mkbinop.pl	Used to create training files for binary addition and subtraction
mkrandomtask.pl	Produces training file for a random problem
visualize_genome.pl	Creates a colorized LaTeXtable showing the contents of a genome file in hex, with comments about the meaning of each block. LaTeXoutput requires the supertabular package.
/training_files	Sample /src/make_train_and_test_set.pl input files
2digitadd.train	addition of two 2-bit binary numbers
binary_counting.train	counting up to seven 'objects' in binary

continued on next page

continued from previous page	
File/Directory Name	Description
rand2.train	
rand3.train	
test1.train	An early test file

Appendix B

Perl Modules

This appendix documents the interfaces of most of the Perl modules used in this project. All the code was developed under Perl 5.8, but should probably run under earlier versions (after 5.005, anyway).

The modules mostly only depend on Perl core modules, with the single exception of NetCompiler.pm, which requires the Template Toolkit (http://www.template-toolkit.org/). They were written on Debian GNU/Linux, but have been used on RedHat Linux, and FreeBSD. It should be easy to get them working on any modern Unix-like system with a recent version of Perl. No guarantees are made.

Hopefully this documentation is complete enough to allow use of the modules, but I am happy to answer question by email. Send any questions to josh@mailsnare.net, but *please know Perl* first.

The documentation for the first module, NetCompiler.pm, begins on the next page.

B.1 NetCompiler

Analyze computational (neural) networks and compile them to C

SYNOPSIS

```
use NetCompiler;
$netcompiler = NetCompiler->new( filename => 'foo.net' );
$netcompiler->compile( 'c', 'foo.c' );
```

DESCRIPTION

NetCompiler is intended to automatically produce C code to execute and train neural networks. It takes as input a specification in one of two formats (see FORMATS); one for humans, one for use in genetic algorithms. It can compile from either of these formats to C code (the primary use) and to input files to graphviz for production of pictures of the network. It cannot recover the input formats from C or graphviz output. See the EXTENDING section for information about extending NetCompiler to handle other formats.

INTERFACE

$netcompiler = NetCompiler->new(<option> => <value> [, ...])

Creates a new instance of NetCompiler. You must specify either the 'filename' option or the 'data' option.

available options

filename

Filename to read network definition from.

data

Scalar or reference of memory structure to read definition from

genome_mode

If set to a true value, input is interpreted as a network genome. (see FORMATS)

$netcompiler->compile(<type> [, <output_file>])

Compiles/translates the network loaded with new() into <type>, which must be one of "graphviz","genome", or "c". If <output_file> is specified, writes compiled network to that file. Returns the compiled network.

FORMATS

The input formats for networks are described in greater detail in the docs/ directory.

Human Readable Format

The human editable format is just a file containing a Perl data structure. It should be an anonymous hash with two entries: OPTIONS and LAYERS, each of which should be a hash ref, with the keys in LAYERS each being a layer name, and the values being arrays of nodes. Each node is an array of input specifiers, and each input specifier is a two element array, where the first element is the name of the node to receive input from, and the second element is the starting weight, or 'R' for random starting weight.

Here is a sample network in this format:

```
{ OPTIONS => { INPUTS => 2,
               OUTPUTS => 1 },
  LAYERS => { A => [ [ ["IN1", 'R'], ["IN2", 'R'] ],
                     [ ["IN1", 'R'], ["IN2", 'R'] ],
                   ],
              OUT => [ [ ["A1", 'R'], ["A2", 'R'] ],
                     ],
            }
}
```

EXTENDING

Extending NetCompiler to have additional output formats is fairly simple. You must create a sub in a separate module which can be passed a reference to the netcompiler object, and an (optional) options hash. There is no formal extension loading mechanism, you must edit the 'compile' sub in NetCompiler.pm, and place the call to your new sub in the if.. elsif.. sequence there.

Extending NetCompiler to new input formats is beyond the scope of this manual. You should become familiar with the source of NetCompiler.pm before attempting this.

Below is documentation of the NetCompiler methods useful for output format modules:

$netcompiler->_input_ids

Returns a list of the IDs of the input nodes.

$netcompiler->_output_ids

Returns a list of the IDs of the output nodes.

$netcompiler->_list_raw_nodes

Returns a list of all node IDs.

$netcompiler->_list_raw_node_ins(<id>)

Returns a list of the IDs of all nodes from which the node with ID <id> receives input.

$netcompiler->_list_raw_node_ins_weighted(<id>)

Returns a list of the inputs for node <id>, with entries of the form: { ID => <input_id>, WEIGHT => <weight> }.

$netcompiler->_list_raw_node_outs(<id>)

Returns a list of the node IDs which receive input from node <id>.

$netcompiler->_list_feed_taints(<$id>)

Returns a list of the feedback group numbers from which the node <id> receives input.

$netcompiler->_<taint_type>_taint(<id>)

Each of the taint functions returns a true value if the node <id> is marked with that taint type. The possible taint types are:

_input_taint
Receives (in)direct input from an input node.

_output_taint
Output from node is (in)directly received by an output node.

_disconnect_taint
Node does not get input from network inputs, or does not send output to network outputs.

_precalc_taint
Node may be computed before any feedback portions of network.

_feedback_taint
Node is part of a feedback loop.

_postcalc_taint
Node is not a feedback node, but must be computed after some feedback nodes.

$netcompiler->_calc_order(<id>)

Returns an integer giving the compute order of the node. Nodes must be computed in ascending order. Nodes with the same compute order are either part of the same feedback loop, or may be computed in parallel.

$netcompiler->_node_taint(<id>, <tainted_by_id>)

Returns true if node <id> receives (in)direct output from <tainted_by_id>.

$netcompiler->_list_node_taints(<id>)

Returns a list of all node IDs from which the node <id> receives direct/indirect input.

$netcompiler->_calc_groups

Returns a list of simultaneously calculable node groupings, in compute order. Each item in the list is a reference to an array of the nodes in the grouping, except for feedback groups, which have all their nodes grouped together in a single sub-array within the compute group.

$netcompiler->opt('optname')

Returns the value of the network option specified by 'optname'. Current option names are 'inputs' and 'outputs', for the number of each in the network.

B.2 Mutation

Mutate binary neural network genomes used by NetCompiler

SYNOPSIS

```
use Mutation;
mutate_genome( <infile>, <outfile>, <mutation_level> );
cross_genomes( <in1>, <in2>, <outfile> [, optional args] );
purge_introns( <infile>, <outfile> );
```

DESCRIPTION

Mutation implements fairly brain-dead mutation and crossover functionality for network genomes. These functions only know that the genetic code is composed of 4-byte blocks.

INTERFACE

mutate_genome(<infile>, <outfile>, <mutation_level>)

Makes an imperfect copy of infile to outfile. The <mutation_level> specifies how many randomly selected data-altering operations to perform per 1000 four-byte blocks. Each operation performed is either a single block deletion, insertion, or replacement with a random value, or a multi-block move operation. The move operation selects a 5-100 block segment and transposes it to a new location within about a thousand blocks of the starting point.

cross_genomes(<in1>, <in2>, <out> [, min [, max [, exponent]]])

Performs a crossover operation, between <in1> and <in2>, and writes the result to <out>. Crossover takes two sequences of 4 byte blocks, and outputs blocks alternating between the two sequences at crossover points. The read position in the 'inactive' file is advanced at the same proportional (to the input file lengths) rate as in the 'active' file. In the following illustration, blocks are represented by letters:

```
<in1>:             AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
<in2>:             BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
crossover points:      *  *      *     *   *     *  *
<outfile>:         AAAABBBAAAAAAABBBBBAAAAABBBBBBAAABBB
```

The parameters min, max, and exponent determine the distribution of segment lengths. Min and max specify the bounds on segment length. The actual length of each segment is found by taking a random number from 0 to 1 (from rand()), raising it to the power of the exponent, and scaling so that 0 becomes min, and 1 becomes max. Higher values of exponent cause more short segments and fewer long segments. The default values of min, max, and exponent are 20, 750, and 3, respectively.

purge_introns(<infile>, <outfile>)

Reads through <infile>, using the start and end markers to keep track of whether the current block is part of a node definition, or not. If it is, it is written to <outfile>. All other (non-coding) blocks are discarded.

B.3 Evolver

Implements a generic genetic algorithm.

SYNOPSIS

```
use Evolver;
$evo = new Evolver( project => 'project_dir',
                    data_log => 'log_name' );
for my $i (0..$number_of_generations) {
  $evo->run_generation();
}
```

DESCRIPTION

Evolver implements a genetic algorithm in a generic form. All particulars of genome encoding, reproduction, selection, fitness functions, etc. are provided in the configuration file. Individuals which are evolved are expected to implement an object oriented interface (see OBJECT INTERFACE).

INTERFACE

Evolver->new(project => <project_dir, [data_log => <logfile>])

Creates a new Evolver object, with the configuration defined in <project_dir>/config. If data_log is specified, lots of generation-by-generation data will be written to <logfile>_<date-time string>.dat, using Storable. See CONFIGURATION for the options in the config file.

$evo->run_generation

Calls fitness function for all networks, does callbacks to select networks to kill, and networks for reproduction, calls reproduction function to produce new networks, and writes data to log file if data_log option was given to new().

OBJECT INTERFACE

The individuals being evolved need to implement the interface defined here. $individual is used for the 'individual' object in this specification.

$individual->save(<project_dir>, <id>)

This individual should be stored under <project_dir> and be retrievable by <id> using the load() sub defined in the config file.

$individual->test_fitness

Should return a numerical fitness value for the individual.

$individual->breed(<mutation_level>, [mate, ...])

$individual should combine with the zero or more mates supplied to produce a single offspring, whose object should be returned.

$individual->kill_files

All files used to store this individual should be purged from the project dir.

CONFIGURATION

The configuration of the genetic algorithm is defined in 'project_dir/config'. This file should be a Perl program which will return a configuration hash when eval()ed. The items in this hash are listed below:

Mandatory Directives

load_individual

Should contain a subroutine reference: load('project_dir', ID), which will return an object representing the individual represented by ID.

parents_per_individual

The number (plus one) of individuals to pass to the breed() function of the 'individual' object. The value 1 corresponds to asexual reproduction, 2 to sexual reproduction, and 3+ to something more exotic.

mutation_level

This parameter is simply passed to the breed() function.

select

Should contain a sub reference: select(<number>, @population). @population will be a list of hashes each having a 'FITNESS' value. The sub should select <number> items from @population and return them.

kill

Should contain a sub reference: kill(@population). @population is the same as for 'select' above. The sub should choose some number of items from the population list, and return them. The number of items in this list will be the same as the <number> parameter passed to the 'select' callback.

get_starting_population

Contains a sub reference which takes as its argument the project dir. Should return a starting population. The size of the starting population is the constant population size for the rest of the experiment.

Optional Directives

average_over

If this option is supplied, it specifies the number of times test_fitness() will be called and the values averaged together to calculate the fitness for new individuals.

rm_killed

If this option is present, and true, kill_files() will be called on all individuals removed from the population.

Other Directives

The 'individual' object methods or other configured subs are also free to include their own configuration directives in the config file. They can access the configuration using the ProjectConfig module, as Evolver.pm does.

B.4 NetEvolvee

Wraps NetCompiler neural networks for use by the Evolver module.

SYNOPSIS

Below is an example config file for Evolver, using NetEvolvee methods.

```
use NetEvolvee;
use Utility "progress_dots";

return { load_individual => \&NetEvolvee::load,
         parents_per_individual => 2,
         average_over => 4,
         mutation_level => 5,
         select => sub { my $nselect = shift;
                         return NetEvolvee::choose_weighted
                           ( $nselect, 10, 5, @_ ); },
         kill => sub { return NetEvolvee::choose_weighted
                         ( 10, 10, 5, 'worst', @_ ); },
         get_starting_population => sub {
           my $project_dir = shift;
           return NetEvolvee::population_explosion( 100, #pop size
                                                    $project_dir,
                                                    'ancestor' );
         },
         introns => 0,
         recompile_genome => 0,
         remove_introns => 0,
         count_partials => 0.15,
         rm_killed => 1,
};
```

DESCRIPTION

This module is not really intended to be used in a script, which explains the unusual format of the SYNOPSIS section.

INTERFACE

$net = load(<project_dir>, <network_id>)

Loads the specified network, which is stored in project_dir/network_id.gen (or .net). Returns a NetEvolvee object.

$net = NetEvolvee->new(object => <obj>, temp => <temp_file>)

$net = $old_net->new(object => <obj>, temp => <temp_file>)

Creates a new NetEvolvee object. If called as a method of an existing NetEvolvee object (as returned by load()), the location of the project directory will be copied from $old_net to $new_net. <obj> should be a NetCompiler object, and <temp_file> should be the filename of the network definition (.gen or .net file).

$net->save(<project_dir>, <net_id>)

Saves the network to <project_dir>/<net_id>.gen. Sets the project dir and id for the $net object, and removes the temporary file in which the network was previously stored. If recompile_genome is set to a true value in the project config, the saved genome file is produced by a call to NetCompiler::compile. Otherwise it is simply copied from the temp file.

$net->kill_files

Removes the files associate with this network.

$net->test_fitness

Calls the train_and_evaluate utility on the network, using the file 'training' in the network directory. Calculates a fitness score between 0 and 2500 based on the results given by train_and_evaluate.

$net->breed(<mutation_level>, [<mate>]);

If <mate> is supplied (it should be a NetEvolvee object), calls Mutation::cross_genomes() on the genome files of it and $net. Either the result of the cross, or the genome of $net is then subject to Mutation::mutate_genome(). If remove_introns is set to a true value in the project config, the network is then processed by Mutation::purge_introns().

select_n_best(<num>, @population)

Assumes @population is sorted by descending fitness, and simply returns a list of the first <num> elements.

choose_weighted(<num>, <multiple>, <max_scale>, ['worst'], @population)

Chooses <num> items from @population by a weighted random selection. Each item in @population should be a hashref, with a key called 'FITNESS'. The item with the highest fitness will be <multiple> times more likely to be selected than the item with the lowest fitness, and the chance for items in between is scaled accordingly. The difference between the lowest and highest will not be exaggerated by more than <max_scale>, however. If the fourth parameter is the string 'worst', the probabilities will be reversed to favor the lowest fitness.

kill_worst([percent => <pct>,] @population)

Returns a the last <pct> percent of @population, which should be sorted in order of descending fitness. If <pct> is not supplied, 10 is used.

population_explosion(<num>, <project_dir>, <parent_id>)

Returns a list of <num> NetEvolvee objects, each of which is created by mutating the network found in project_dir/networks/parent_id.net (or .gen), with a mutation level of 10.

B.5 ProjectConfig

Provides a cached loading mechanism for Evolver and NetEvolvee.

SYNOPSIS

```
use ProjectConfig;

$config = ProjectConfig::get_config( 'project_dir' );
```

DESCRIPTION

This package only provides one function: get_config, which loads the file 'project_dir/config', evaluates it, and returns the return value from eval(). The return values are cached, so that on subsequent calls with the same value of 'project_dir', the config file will not be re-read unless it has changed on disk. Also makes sure that the value returned by config is a hash reference which contains the keys 'load_individual', 'parents_per_individual', 'select', 'kill', 'get_starting_population', and 'mutation_level'. If not, it will raise a fatal error with die().

B.6 Utility

Provides two utility functions

SYNOPSIS

```
use Utility qw(progress_dots maketemp);

print "Doing something"
for my $i (1..$n) {
  do_something();
  progress_dots( 50, $i, $n );
}
print "done\n"

$temp_file = maketemp();
```

DESCRIPTION

This package provides two utility functions which are used by Evolver.pm and NetEvolvee.pm

INTERFACE

progress_dots(<num_dots>, <num_completed>, <num_total>);

Prints period characters "." to STDOUT as a text-only progress bar. Should be called with steadily increasing values of <num_completed>; if this value decreases from one call to the next the progress "bar" is restarted. Remembers the previous value of <num_completed> and prints an appropriate number of dots. If the value of <num_completed> actually reaches <num_total>, exactly <num_dots> will have been printed.

$tempfile = maketemp()

Tries to provide a uniform interface to the mktemp command across several operating systems, so that mktemp will respond to the 'TMPDIR' environment variable. It is known to work on Debian 'sid', and versions of Redhat, and FreeBSD. It should be tested before use to ensure proper operation. Returns the path to the temporary file created.

Appendix C

Neural Network Utility Library

This appendix documents the set of convenience wrappers around the neural network libraries produced by the NetComplier modules (section B.1). This documentation is fairly minimal, so perusal of the source code is recommended and a good command of the C programming language is essential. Questions can be directed to josh@mailsnare.net.

All C code in this project was developed on Debian GNU/Linux 'sid', and has been used successfully on RedHat Linux, and FreeBSD. GNU make is needed to build the projects, and the code has only been compiled using GCC. It should be portable to most modern Unix operating systems with a GNU toolchain, though YMMV. It probably could be ported to anything POSIX compliant. Note that Windows is not really POSIX compliant — you are on your own if you want to port it to Windows; *do not* ask me for help with this.

Functions with `int` return values return 0 for success and −1 for failure, unless otherwise noted. There are several types used in the functions below which are specific to this library, namely `net_definition`, `net_io`, and `net_weights`. These are just `typedef`'ed `structs`, but for the purposes of the main interface, they can be treated as opaque values. They are documented in Section C.3.2.

Your program must `#include "neural.h"` in order to use these functions.

C.1 Main Interface

`char` ***neural_error** ( );

Returns a pointer to the libneural error string buffer, which will contain a description of the error which occurred in the last function call. This is a statically allocated buffer which can be overwritten at any time. The contents should be copied if they are needed after the next call to a libneural function.

`int` **load_net** (const char *net_file, net_definition *def);

Loads a compiled neural network shared library. The argument `net_file` should contain the name of the .so file to load, and `def` should point to an already allocated `net_definition`, which will be set with the parameters of the loaded network.

```
int init_net_io ( net_definition *def,
                  net_io *io,
                  int with_internal_state );
```

Allocates and initializes the input and output buffers in `io` according to the network parameters in `def`. If `with_internal_state` is true, space to store the output states of all nodes in the network will also be allocated in `io`.

`int` **init_net_weights** (net_definition *def, net_weights *weights);

Allocates and zeroes space for network weights in `weights`, appropriate to the network defined by `def`.

`int` **starting_weights** (net_definition *def, net_weights *weights);

Allocates space for weights, like `init_net_weights`, and populates the weights with the network's internal starting weights.

`void` **free_net_io** (net_io *io);

Frees buffer space in `io`. Care should be taken not to use `io` after this function has been called.

`void` **free_net_weights** (net_weights *weights);

Frees buffer space in `weights`. Do not use after this function call.

`int` **copy_net_io** (net_io *dest, net_io *src, int copy);

Copies values from `src` to `dest`. Which parts of the information are copied is determined by `copy`, which should use the constants `COPY_INPUT`, `COPY_OUTPUT`, or `COPY_INTERNAL`. These can be OR'ed together, and the constant `COPY_ALL` is provided as a convenience.

`int` **test_io_output** (net_io *a, net_io *b);

Tests whether the output portions of `a` and `b` match. If all the outputs are the same, returns 1. Otherwise, returns -1 times the number of matching outputs.

`void` **apply_weights** (net_weights *weights, net_weights *changes);

Adds the values in `changes` to `weights`. This can be used, for instance, when `train_net()` is called without the `NET_APPLY_WEIGHT_CHANGES` flag.

`int` **calc_net** (net_definition *def, net_io *io, net_weights
*weights);

Calculates the outputs from net `def` using the inputs from `io` and the `weights`. Outputs are written to `io`, and if it was initialized with an internal state buffer, the internal output values of nodes in the network are written to `io` as well.

`void` **train_net** (net_definition *def, net_io *io,
net_weights *weights,
net_weights *weight_changes,
double training_level,
unsigned int flags);

Trains the network. `io` should contain internal states from a call to `calc_net`. The value of `training_level` is the constant η used to scale the final value of each weight change. A value of 0.1 works pretty well. What is done with `weights` and `weight_changes` depends on the value of `flags`. There are 3 constants available, which can be combined with `||`.

`NET_CORRECT_OUTPUTS_GIVEN` Indicates that `io` contains the correct outputs, which are then used for standard back propagation training. The correct outputs can be written into `io` after a call to `calc_net` using `copy_net_io`. If correct outputs are not supplied, the neural network tries to perform a reinforcement-type training, which is a feature of the network code produced by NetCompiler.pm, and not of libneural, and it may not in fact be working properly at the time of this writing.

`NET_APPLY_WEIGHT_CHANGES` If this flag is specified, the weight changes calculated will be added to `weights`.

`NET_ACCUMULATE_WEIGHT_CHANGES` If this flag is given, `weight_changes` will not be overwritten with new values, and instead the deltas will be added to the existing values in `weight_changes`. If this option is used in combination with `NET_APPLY_WEIGHT_CHANGES`, the deltas will be added to both weight sets.

`int` **fwrite_weights** (FILE *file, net_weights *weights);

`int` **fread_weights** (FILE *file, net_weights *weights);

`int` **fwrite_net_io** (FILE *file, net_io *io);

`int` **fread_net_io** (FILE *file, net_io *io);

These functions operate on already open `FILE *` file-handles, as used by the standard library functions `fopen()`, `fwrite()`, `fread()`, etc. The format is simple: values are printed in ASCII, one to a line, with the number of values preceding the values themselves. The net io functions begin with the number of input, output, and internal values (which can be zero) in that order, and then list the values themselves in the same order. When reading, there should be the same number of data in the file as there are 'spaces' in the structure being populated, or an error will result, except when `fread_net_io` is passed a `net_io` which was initialized without internal state buffers, in which case the internal state values in the file will be ignored if present.

The author has found it easiest to create input and output patterns manually in a file or using a short Perl script and use these functions to load them in, rather than populating the `net_io` structure manually in a `C` program. See `make_train_and_test_set.pl` for an example of how to do this.

C.2 Pattern Sets

Two functions are provided for operating on sets of input/output patterns. These sets are just arrays of pointers to `net_io` structures. Correct outputs are assumed to be supplied in the sets.

`int` **fread_net_io_set** (FILE *file,
net_io **set_buf, net_io ***set, int *count,
net_definition *def, int with_internal_state);

This function is similar to `fread_net_io()`, but reads a list of `net_io` variables at once. The file should contain the number of `net_io` at the current read position, and the data for all of them immediately following. `set_buf` should point to a variable of type `net_io *`. This will be populated with the location of the memory segment allocated for the `net_io` structs. `set` should point to a variable of type `net_io **`. It will be filled with the location of the pointer array for the set. `def` and `with_internal_state` are passed internally to `init_net_io`. If `with_internal_state` is nonzero but the file does not contain internal state information, an error will result.

```
typedef struct {
  int    iteration_count;
  int    presentation_count;
  int    training_count;
  int    correct_count;
  int    learned_count;
  float  elapsed_seconds;
  float  correct_rate;
  float  learned_fraction;
} training_statistics;
```

`void` **train_on_set** (net_definition *def,
net_io **training_set, int set_count,
net_weights *weights,
double training_level,
double timeout_secs,
training_statistics *stats,
int flags);

This function iterates through the io patterns in `training_set` until either the network produces correct output for all patterns in a single iteration, or `timeout_secs` seconds elapse. The only flag available is `TRAIN_ON_SUCCESS`, which causes the network to be trained even if it produces the correct output - this will reinforce the correct outputs somewhat. The other parameters are the same as for `train_net()`, but `weights` will always be modified to contain the trained weights.

Statistics about the training process are placed into `stats`. The meanings of the values are as follows:

`iteration_count` - Number of iterations through the training set.
`presentation_count` - Number of times input patterns are presented to the network.
`training_count` - Number of times the network is trained.
`correct_count` - Number of times the network produces correct output.
`learned_count` - Number of patterns successfully learned by the network at the final iteration.
`elapsed_seconds` - Amount of time elapsed during training.
`correct_rate` - Overall fraction of presentations which elicited correct output.
`learned_fraction` - Fraction of the set successfully learned.

```
typedef struct {
  int    successful_items;
  float  success_rate;
  float  partial_success_avg;
} test_statistics;
```

`void` **test_on_set** (net_definition *def
net_io **test_set, int set_count,
net_weights *weights,
test_statistics *stats,
int flags);

Presents each pattern in the set to the network once and records statistics about the correctness of the outputs. The values of `weights` should probably have been filled in by some sort of training, such as that performed by `train_on_set()`. The value of `flags` is ignored, it is present simply to allow behavior altering flags to be defined without altering the function prototype. Here are the meanings of the values in the `test_statistics` structure:

`successful_items` - Number of patterns for which output was correct.
`success_rate` - Fraction of the set correct.
`partial_success_avg` - Average fraction of correct outputs for all incorrect patterns.

C.3 Internals

This section documents the interface which the neural networks used by this library must support and the 'opaque' structures used by the interface documented in Section C.1.

C.3.1 Network Module Interface

In order to be used by this library, a network must be contained in a dynamically loadable library (a file with the extension ".so" on most Unix-like operating systems). It must also provide the functions documented below.

All output values are expected to range from -1 to $+1$. The various 'count' parameters are provided to allow consistency checking, and may be ignored if you choose to trust the calling code. Ordering of the values in parameter arrays is up to the network code, but should be consistent across all functions.

```
struct net_info {
  int  input_count;
  int  weight_count;
  int  output_count;
  int  node_count;
};
```

`void` **_net_info** (struct net_info *info);

This function should populate the `info` struct with the number of inputs, weights, outputs, and nodes in the network.

`int` **_setup_initial_weights** (double *weights, int weight_count,
unsigned int *seed,
int make_seed);

Should fill the `weights` array with starting weights. If some of the weights are random, should use the value pointed to by `seed` to seed the random number generator, for repeatability. If `make_seed` is true, the function should generate a random seed and place it in `*seed`. If no random weights are generated, or `seed` is null, `seed` and `make_seed` can be ignored.

`int` **_calc_net** (int *inputs, int input_count,
int *outputs, int output_count,
double *weights, int weight_count,
double *node_values, int node_count,
int feedback_limit,
double feedback_convergence);

Should calculate the output values produced by the network from the supplied inputs when connections are weighted with the values in `weights`. Output values should be placed in `outputs`. If `node_values` is not null, floating point output values for each node should be placed into it. `feedback_limit` specifies the number of iterations allowed for any feedback portions of the network to 'settle', and `feedback_convergence` is the threshold for the amount of change in the absolute value of a node's output between iterations below which the node should be considered to have a 'steady' value.

`void` **_train_net** (double *weights,
double *weight_changes, int weight_count,
double *node_values, int node_count,
int *correct_outputs, int output_count,
int feedback_limit,
double feedback_convergence,
double training_level);

Should train the network. Most of the parameters are the same as for `_calc_net()`, with a few exceptions. `node_values` is provided to obviate the need to recalculate the network, and is not intended to be overwritten. The changes in the network weights should not be applied to `weights`, but should be written into `weight_changes`. `training_level` should be used to scale these values.

The network may choose to implement a reinforcement-type training (no correct outputs are provided). This mode should be used when `correct_outputs` is null. A negative value for `training_level` should be interpreted as negative reinforcement ('punishment').

C.3.2 Opaque Interface Types

```
typedef struct {
  calc_network_fn            calculate;
  setup_initial_weights_fn   setup_weights;
  train_net_fn               train;
  net_info_fn                get_info;
  struct net_info            info;
  void                      *dlref;
  int                        feedback_limit;
  double                     feedback_convergence;
} net_definition;
```

The first four members of this structure are pointers to the network module functions (Section C.3.1). `info` contains the values provided by the info function in the network. `dlref` contains the value returned by the `dlopen()` system call. The values for `feedback_limit` and `feedback_convergence` are passed to the network's calculation and training functions. `load_net()` initializes these to 1000 and 0.05, respectively, but they can be changed subsequent to that call.

```
typedef struct {
  int     *inputs;
  int      input_count;
  int     *outputs;
  int      output_count;
  double  *node_values;
  int      node_count;
} net_io;

typedef struct {
  double   *weights;
  int       weight_count;
} net_weights;
```

The values in these two structures should be self-explanatory. These structures should only be used after the appropriate "init" function is used. Remember that `init_net_io()` may leave `node_values` null, and be sure to check for that condition before attempting to write to it.

Bibliography

[1] M. Archetti. Loss of complementation and the logic of two-step meiosis. *Journal of Evolutionary Biology*, 17(5), Sep 2004.

[2] Theodore P. Hoehn. Wolves in sheep's clothing? the effects of "hidden" parental mutation on genetic algorithm performance. In *Proceedings of the 36th annual Southeast regional conference*, pages 221–227. ACM Press, 1998.

[3] Richard G. Palmer John Hertz, Anders Krogh. *Introduction to the Theory of Neural Computation*. Santa Fe Institute Studies in the Sciences of Complexity. Perseus Books Publishing, 1991.

[4] Bryant A. Julstrom. Seeding the population: improved performance in a genetic algorithm for the rectilinear steiner problem. In *Proceedings of the 1994 ACM symposium on Applied computing*, pages 222 – 226. ACM Press, 1994.

[5] Aaron Konstam Michael Jones. The use of genetic algorithms and neural networks to investigate the Baldwin effect. In *Proceedings of the 1999 ACM symposium on Applied computing*, pages 275–279. ACM Press, 1999.