GANN - The Manual

Contact: r.beiko@imb.uq.edu.au
Last modification: Documentation updated May 2004.
GANN Homepage: http://bioinformatics.org.au/***

Contents

Introduction

The GANN software suite was developed mainly to enable the extraction, conversion, and comparison of specific regions of DNA within the genome. The important steps in this analysis are:

- Extract sequence and open reading frame (ORF) information from files, identify different regions of sequence relative to the start and stop codons (coding sequence, proximal upstream sequence, distal upstream sequence, non-upstream sequence).
- Split these different types of sequence into smaller 'windows', and convert these windows into a set of numbers (or 'indices') that represent different sequence and structural properties of the DNA.
- Train populations of feed-forward multi-layer perceptrons (MLPs) to make functional predictions about a specific sequence based on the indices derived from it.
- Collect and summarize the mountain of data generated by the neural-network analysis, and compare these results to similar results from different positive and negative controls.

At first, the entire software was contained within a single quasi-C++ program, but the need for generality and flexibility within the analysis convinced me to spin off the different steps of the analysis into separate programs. The programs in the suite are chained together such that the output from one serves as the input to the next, but the file formats are very simple and straightforward, so you could bring the output from 'indices' into a spreadsheet program for statistical analysis, or use your own data set as input to 'GANN', for instance.

Significant changes have been made to GANN since version 1.0, mostly dealing with the representation of data. Please consult the Change Log at the end of this file for more information.

Extending GANN

The GANN suite is designed with flexibility and some degree of extensibility in mind. The GANN program, for instance, implements two different MLP training algorithms (Backpropagation and Genetic Algorithm), and the majority of training parameters can be specified by the user or optimized within the program. However, the object-oriented implementation of MLP should make it easy (or at least 'easier') to implement other MLP training algorithms, since many of the existing MLP functions will not need to be rewritten.

The other programs in the suite make far less use of non-STL C++ objects, and it may be appropriate to convert some of the ideas in 'indices.cpp', for instance, into proper classes. This would make it far easier to extend the types of indices that can be generated. Some other ideas for modifying the software and/or playing with the data will be described in the appropriate sections.

I would like to hear about user extensions and modifications to GANN, as well as suggestions for improvement and reports of bugs or strange behaviour.

Compiling and Running GANN

GANN has an associated makefile that should compile and link the appropriate .cpp files (for UNIX systems) and create a series of directories. The default compiler specified in the makefile is GNU G++, but you can change this to CC or another compiler if you prefer. For Windows systems, you can use your favourite compiler to create the appropriate executables, or just download the executables and arrange them however you like in a directory tree.

Unless you are using giant data sets (see below for definition of a giant data set) most of the programs in the GANN suite will run in a few seconds or a couple of minutes. The exception is the GANN program proper, which can take a very long (days, weeks) time to run if you want lots of optimization rounds and large data structures. If you plan to use GANN with large datasets, I suggest finding a compiler that will do the best job in optimizing your code as it can make a substantial difference in execution time.

The Perl scripts in this suite should be run with a Perl 5.x interpreter.

License Information

Acknowledgments

A number of ideas implemented in this software were contributed by my Ph.D. supervisor, Dr. Robert L. Charlebois of NeuroGadgets, Inc. and Adjunct Professor at the University of Ottawa. The design and source code for the genetic algorithm classes Chromosome, Evolvable and Selectable are modified from versions created by Dr. Charlebois.

Other important ideas were contributed by the members of my supervisory committee: Dr. Donal Hickey and Dr. Alain St-Amant of the University of Ottawa, and Dr. Iain Lambert of Carleton University.

Program Descriptions

GETSEQ.pl

Usage: perl GetSeq.pl (-g gbkfile | -n NGIBWSfile rawfile) outfile seqlen [-C xxxx] [-U xxxx] [-F xxxx] [-N xxxx]

GetSeq.pl takes sequence data and predicted ORFs in either of two formats:

1) NCBI .gbk format genome file, with predicted ORFs and raw sequence at the end.
2) A table of predicted ORFs from the NeuroGadgets Inc. Bioinformatics Website (www.neurogadgets.com). This list is generated by displaying the ORF starting position within the ‘Sorted list of protein characteristics’ query,

and displaying the ORF end point as a secondary characteristic. If this format is specified, a FASTA-format file containing the raw genomic DNA sequence must also be specified.

GetSeq.pl parses the ORF file, and builds a list containing the names and start/stop positions of the ORFs in the genomic sequence. The DNA sequence is then partitioned into different classes, based on its position relative to the ORF start and stop points. These classes are:

1) Coding sequence – Sequence located between the start and stop points of the ORF.
2) Upstream intergenic sequence – The first *n* bases upstream of the start codon.
3) 'Far upstream' intergenic sequence – A block of DNA sequence found entirely > *n* basepairs upstream of the start codon.
4) 'Non upstream' intergenic sequence – Sequence found between two convergently transcribed ORFs, and therefore is downstream of both of them.

From each of these sequence categories, a user-specified number of sequences of length *n* will be extracted. If the number of available sequences of length *n* is less than the requested number, GetSeq will output as many as it can extract. In either case, GetSeq reports the total number of sequences extracted for each category.

Output: The output is a three-column comma-delimited file. The first column represents the set category of this entry: Upstream sequences are assigned to category 1, while coding sequence, and far upstream / non upstream sequences are assigned to category 0.

The second column is a string, the first letter describing the class of the sequence (C, U, F, or N). For the categories associated with a specific ORF (C and U), this letter will be followed by the gene name and product name from the GenBank file (if applicable). For categories F and N, the first letter will be followed by a number that gives the approximate location (within about 500 nucleotides) of that sequence within the original genome file.

The final column is the extracted DNA sequence of length *n*. This will represent the 'sense' strand of coding and upstream regions, while for categories F and N, the sequence from the raw DNA file will be converted to its reverse complement half of the time.

Notes on ORF designations: The GenBank-format genome file can contain information about non-protein-coding sequences, including pseudogenes, RNA genes, and introns. They are handled as follows:

Pseudogenes are sometimes identified with the identifier '/pseudo' in the .gbk file. These sequences are treated as genes when sequences are partitioned into four categories, but they will NOT be added to sets C or U and will not be found in the output file.

RNA genes are treated like regular genes. Their sequences may therefore be found in the output file. If you like, you can delete entire lines from the output file and still use the file as input to the next program (Indices).

Introns, as highlighted by the 'join' identifier in the GenBank file, are currently ignored. This functionality (which is somewhat important for eukaryotic analyses) will hopefully be implemented later.

Extracted sequences that contain IUPAC degenerate base codes will not be added to the output list.

INDICES

Usage: indices seqfile outfile win-size overlap [-m mapfile] [-o oligofile] [-n num-nmers] [-i num-nucints] [-r numReps]

'Indices' converts extracted upstream, within-ORF, non-upstream and far upstream sequences into a set of numbers. The input sequences should be in the format generated by GetSeq.pl.

The sequences in the input file will be split into 'windows' of size *win-size*, with an overlap of size *overlap*. Each window will be assigned a number, with the window at the 3' (i.e., the 'right') end of the sequence assigned the number 0, and window numbers increasing in the 5' direction. For upstream sequences, this means that window 0 is closest to the start codon.

Four different types of nucleotide conversion rules are implemented:

- Nucleotide mapping rules, where each n-mer is assigned a user-specified floating-point value. The total value of a sequence will then be the sum of all its constituent n-mers.
- Counts of specific oligonucleotides, provided by the user in a separate file, one per line. A hoped-for extension would allow the calculation of standard statistics of nucleotide over- and under-representation.
- Counts of all non-degenerate n-mers up to a specified length.
- Counts of all nucleotide intervals of the form $X(N)_pY$, up to a user-specified *p*.

In the mapping rules and oligonucleotide counts, the user can specify degenerate sequences. However, the program will exit if a given map definition does not account for all possible non-degenerate n-mers.

Input map format: As follows:
{ NAME SIZE { OLIGO1 VAL1 OLIGO2 VAL2 … OLIGON VALN } }
NAME = Name of the mapping rule

SIZE = Size of the oligos specified by this rule
OLIGO1 VAL1 = An oligonucleotide (degenerate or nondegenerate) followed by the numeric value to be associated with it

Any combination of degenerate and non-degenerate (IUPAC) bases can be specified, but each map must specify EVERY possible word of size *n*. For instance, if the word size is two, then all of { AA, AC, AG, …, TT } must have associated rules. A definition that included { RR, RY, YR, YY } would be acceptable. Oligo names and associated values can be separated by any type of whitespace (including newline) or punctuation (except '{' and '}'), which allows custom formatting of the map file for readability.

<u>Input oligonucleotide format</u>: One oligonucleotide per line. Again, degenerate nucleotide codes are accepted, so you could count all instances of 'ACNNNNNNNRDT' in a sequence.

<u>Complete n-mer and nucleotide interval counts</u>: Specified on the command line. 'indices' will save a number of different files, each beginning with the 'outfile' argument specified at the command line. The indices that result from each map calculation will be output to a separate file, with the map name included in the filename. The oligos will all be output to a single file ending in OLIGOS. For the 'nuc-int' and 'nmer' categories, each separate word size will be output to a separate file.

Number of replicates: The final command-line option allows the user to calculate a Z-score based on a number of shuffled versions of the original windows of sequence. For instance, if numReps = 100, then each window of sequence will be shuffled 100 times, and all of the index values will be recalculated for each replicate. The Z-score is calculated as:

Z = ('real' score (mean score of replicates)) / (standard deviation of replicates)

COMBINE

Usage: combine outfile [-g] [-noZ] [-w minWin maxWin] [-e numExp] [-f numNeg] [-h filehandle]

'combine' will read a set of index files from the current directory, and combine them into a format readable by GANN. This program is interactive, and once run requires the user to accept or reject a list of files to be combined. If a filehandle is specified, then the list of files for selection will be restricted to files containing the filehandle. The appearance of a filename in the interactive list does NOT mean that Combine can necessarily read it, but Combine will let you know later on in the execution phase that a given file cannot be processed. Subdirectories will not appear in the selection list, whether a filehandle is specified or not.

The list of sequences MUST be identical in each of the input files. If the files were all generated from the same initial set (i.e., the output of a GetSeq analysis), then the sequence lists *will* be identical. This condition is not checked by Combine.

In contrast with the index files produced by Indices which begin with the list of index names, Combine will save its files with two additional header lines. The first line is a single number that reports the number of indices, and the second shows the membership of each set in the order (set 0 train, set 1 train, set 0 test, set 1 test). Combine can take its own output files as input, which allows hierarchical Combine jobs.

Options

If requested with the -g option, Combine will also calculate Global values for each group of windows in the input files. A group of windows is a set whose names differ only in the window number identifier. The global values represent the maximum, minimum and average values of all windows in a specified group. These calculations will be performed separately on window values and on Z-scores, since the two will not typically have the same scale.

If you do not wish to include the Z-scores in the output file, the -noZ option will suppress every index name ending in 'Z'.

Similarly, you can tell the program to only include a specific range of windows with the -w option. If you specify a minimum and maximum window number, only those windows will be output in the combined file (and considered for -g analysis). Out-of-bounds window values will be replaced by the extreme values in the input file.

Output Files

There are two options for output files, '-e' for experimental output and '-f' for negative control output. The program defaults to '-e 1', which will produce a single output file with each line randomly assigned to a training or test set case but with positive / negative set membership untouched. If a larger value for -e is specified, then multiple output files will be produced, with different train/test assignments for each file. If '-f' is specified, then the requested number of negative control files will also be produced, with random train / test set assignments AND random reassignment of positive / negative set members. The '-f' flag overrides the '-e 1' default, and no experimental output files will be generated if '-f' is used without '-e'.

DEFINEVARS

Usage: DefineVars

DefineVars is an interactive program that allows you to generate a 'command file' that specifies the global variable values that GANN will use. The program is not strictly necessary for two reasons:

1) The command file is a plain-text file, and can be generated with any text editor.
2) GANN will apply its own set of default variable values if it can't read some (or all) of them from the command file.

That being said, DefineVars is strongly recommended as a tool, because the command file is long (currently defining ~65 variable values, some with quite complicated names), and because you will NEVER want to use the entire set of default variable values for a GANN run.

DefineVars presents the user with a main menu, from which they can select among options that represent the different sub-groupings of variables. This menu is case-insensitive. Choosing a category will bring up a sub-menu, with numbered options to set different variable values. The program does not check the type of the variable entered, so you can enter whatever you like and the program will accept it. Most variables are either integers or floating-point values (including Boolean true-false values implemented as floating-point type). See the 'GANN Variables' section below for full details.

The final two options in the main menu allow you to save the variables to an output file, and to quit the program. Any variables that have not been explicitly set will retain their default values. Quitting will NOT save the variable values to a file.

GANN (Core)

Usage: GANN (-g | -b) indexFile commandFile **OR** GANN –l trainedNet indexFile

The GANN program performs the neural network training, and optimizes the input indices and neural network parameters. To perform rounds of training and optimization, GANN requires two input files: the command file generated by DefineVars, and an index file generated by Combine. GANN can also take a trained neural network and index file as input, and output the neural network predictions for the sequences represented in the index file.

The type of neural network to be trained is specified by the '-g' or '-b' option: '-g' will set up and train populations of genetic algorithm-trained neural networks, while '-b' will create populations of backpropagation-trained neural networks. Each of these neural networks will then be trained with a subset of indices.

The core idea of GANN is the use of genetic algorithms to optimize parameters. Briefly, a set of parameter values used to define a system (the 'genes') is grouped together into a Chromosome object. A set of chromosomes, each with different values for the set of parameters, are collected into populations called Evolvables, and these populations can be collected into groups of populations called Selectables. Each Chromosome within a population specifies an instance of a given data structure, and a round of training is performed by training and testing all of these data structures. At the end of the round, all of these data structures will have an associated score, which is treated as the fitness of the parameters specified by each Chromosome. Evolutionary operations can then be performed using the fitness scores:

- RECOMBINATION can occur between two Chromosomes with high fitness scores, yielding an offspring Chromosome with some parameters inherited from both parents. Recombination can only occur between members of the same Evolvable population.
- MUTATION can modify the parameter values specified by a single Chromosome.
- MIGRATION can transfer Chromosomes between separate Evolvable populations, which are otherwise reproductively isolated.
- ELIMINATION (‘killing’) of pairs of Chromosomes that are very similar, to avoid homogenization of a population. One or the other of a given pair of Chromosomes will be killed in this case.

The genetic algorithm is implemented twice in the software. The Outer Genetic Algorithm (OGA) is used to optimize neural network parameters, and the choice of input indices to present to the neural network. An OGA Chromosome specifies a complete set of neural network parameter values, as well as a user-specified number of indices to present as input. The OGA Chromosome is used to create either a single neural network (if backpropagation is used) or a population of neural networks (if genetic algorithm neural networks are used). The fitness of an OGA Chromosome is determined using accuracy scores on the *test set*, so that OGA evolution is based on generalization ability.

If the ‘-g’ option is used to specify genetic algorithm neural networks, then an Inner Genetic Algorithm (IGA) will be created by each OGA Chromosome. Each *IGA* Chromosome will specify a complete set of link values for a neural network. A round of IGA training will score each of these neural networks on the *training set*, then the connection weights will recombine, mutate, etc. to yield the next population of IGA Chromosomes. In a final round of IGA training, the best *test* scores will be recorded and used to determine the fitness of the relevant OGA Chromosome.

If the ‘-b’ is used, then either a single backpropagation network or a small number of non-interacting ‘replicates’ will be created by each OGA Chromosome, and each of these will be trained and tested separately using the backpropagation algorithm. At the end of a specified number of runs, or if certain premature stopping conditions are met, then the scores on the *test set* will be used as the fitness of the relevant OGA Chromosome, as above.

Two methods of scoring the training and test sets are used: a ‘false positive’ (FP) score and a distance score. Both are reported in output files (four of them, named nn(train|test)(fp|dist)scores.csv), but only the FP score is used for selection and training. The FP score is calculated differently for single versus multiple outputs.

For single inputs, the neural network's prediction is considered 'positive' if the output node value is greater than 0.0, and 'negative' if the prediction is less than or equal to 0.0. A score of 1 is assigned for each correct prediction, and 0 for each incorrect prediction. The FP score is then the average of these scores. For multiple output nodes, a score of 1 is assigned if the largest prediction score is found at the output node corresponding to the correct group, and 0 if the largest score is not at the correct output node.

The distance score is simply the absolute value of the difference between the observed (prediction value at a given output node) and expected (1.0 is the output node corresponds to the correct set, -1.0 otherwise) values. These values are again averaged out for all predictions on the training or test set.

GANN also produces a file called 'nnparams.csv' that shows the neural network parameters, best scores and inputs used by each OGA Chromosome. This file is essential to the execution of 'collect.pl' below.

COLLECT

Usage: perl collect.pl indexFile scoreFile round

A neural network analysis / optimization run carried out by GANN will typically produce a giant amount of output. While the scores, inputs and parameters can be assessed using many different techniques, collect.pl will calculate some generally useful summary scores and statistics.

IndexFile is simply the index file that was presented to the GANN core software. ScoreFile is the GANN output file to be analyzed, one of 'nn(train|test)(fp|dist)scores.csv', and 'round' is the round of OGA training to be reported (usually the final round, equal to the total number of OGA training rounds). This program also needs the file 'nnparams.csv' which is also produced by GANN.

Collect.pl will produce three output files:

- input-sum.csv shows the number of times each index is present in each OGA training round. This allows the increase or decrease in frequency of different indices to be tracked and compared.
- paramSum.csv is a summary of the neural network parameters for each training round. The variables that are subject to selection by the OGA (number of hidden nodes, etc.) are summarized, showing their maximum, minimum and mean for each training round, as well as the standard deviation of the mean.

- scoresum.csv is a summary of the OGA chromosome scores associated with each index in the training round specified on the command line. The mean, maximum, and minimum OGA chromosome scores for each index are shown, followed by a list of all scores for chromosomes containing that index. This list can then be sorted on any of the max, min, or mean values, and the list of the best-scoring indices thus obtained can be presented to ‘whittle.pl’ (see below).

WHITTLE

Usage: perl whittle.pl infile IDs outfile

Whittle is a useful little program that will reduce the number of indices in an index file. 'Infile' is the input index file in post-combine format (with two lines of header information), 'outfile' is the name of the new file with fewer indices, and 'IDs' are the range of indices to keep in the new file. The first index is identified as '0'.

If you wish to keep a contiguous set of indices (say, from the first to the two hundredth), then it is sufficient to enter '0-199' in place of 'IDs'. Entries such as '-199' and '50-' are also acceptable, and the first and last index positions will be substituted in, respectively. '-' is also acceptable but useless, since it saves every index from the first to the last in the new file.

To extract a non-contiguous set of indices, you can list the index numbers (remember that 0 is the first index) in a text file, one per line. The filename would then be the argument to use in the 'IDs' position. The sorted list from 'Collect' above is a good example of this, if you wanted to extract the best-scoring indices from the larger set. The indices need not be sorted.

GANN VARIABLES

The list of GANN variables is quite long, and there are some subtle (and not so subtle) interactions that can occur between them. The most important of these interactions are outlined below.

There are currently four different data types associated with different variables. It is important to adhere to these types, or certain variable values may be assigned surprising values when the GANN runs are carried out. The data types are:

Integer: An integer value, such as ‘1’ or ‘6’.
Floating-point: A floating-point value, such as ‘0.01’ or ‘2.0’.
Text string: A sequence of letters, numbers, and/or punctuation, such as “ogastats.csv” or “outfile1.txt”.
Pseudo-boolean: A condition that can have the value ‘true’ or ‘false’. In the command file, these values are represented as floating-point values in the range [0.0, 2.0) where any

value less than 1 is interpreted as ‘false’, and any value of 1 or greater is ‘true’. Where a range of values must be specified, this allows the user to choose the starting proportion of OGA chromosomes that will be ‘true’ or ‘false’ for a given characteristic. If the minimum specified value is 0.000 and the maximum is 1.999, then the proportions should be roughly equal.

Some variables can be overridden or made redundant by the values of other variables. The most important examples of this effect are the GARANGE and BPRANGE variables. If ‘-b’ is specified on the GANN command line, then the GARANGE variables will not be used. Similarly, using the ‘-g’ option will cause the program to ignore the BPRANGE variables. Other such variables are indicated with a leading * (see below).

There are a handful of redundant variables (identified with parentheses) whose definitions have been maintained. Changing the values of these variables will have NO impact on the analysis. These variables may be reactivated in later versions of the software.

Finally, problems can arise from interactions between the number of Chromosomes in an OGA or IGA population, and the parameters that guide their evolution. For instance, if a given population only has five Chromosomes, and the proportion of chromosomes permitted to recombine is only 0.1, then the program will get stuck because it will not be able to select two parents to recombine. Be aware of these combinations, and try to use tens or hundreds of Chromosomes rather than a few.

Category NNRUNS (3 members)

NN_TRAIN_RUNS: Number of neural network training rounds to perform. **Integer** > 0.

*REPLICATES (BP only): Number of times to replicate neural network training and testing (i.e., starting over with a new random set of connection weights.) GANN will set this value to 1 if GA runs are being performed. **Integer** > 0.

*NUM_TO_RECORD (GA only): How many scores from each population of IGA chromosomes to record for testing purposes. The score assigned to a given OGA chromosome will be the average of the best NUM_TO_RECORD neural nets within the IGA population. **Integer** > 0.

Category FILENAME (2 members)

NET_DEF: The filename used to save all trained neural networks with a generalization score above MIN_GEN. **Text string**; should contain legal filename characters.

OGA_DEF: A small 'progress report' file that outputs a single line for each OGA training round, along with the best training and test scores obtained to date. **Text string**; should contain legal filename characters.

Category SCOREPARAM (4 members)

WORST_SCORE: When using generalization scores for selection, a constant equal to WORST_SCORE is first subtracted and the resulting generalization score is rescaled between 0.0 and 1.0. Any generalization score that is equal to or worse than WORST_SCORE will be assigned a score of zero. This allows the user to label any score less than a given minimum (e.g., 0.5) as equally bad. **Floating-point** value between 0.0 and 1.0, where 1.0 would serve as an interesting negative control (all generalization scores are equivalent).

MIN_GEN: The lowest generalization score required for a given neural network to be saved in the NET_DEF file. **Floating-point** value between 0.0 and 1.0, but setting MIN_GEN to less than about 0.7 will yield a giant output file for many problems.

IVO: Input vector optimization on or off. This determines whether the neural network inputs are optimized by the OGA. **Pseudo-boolean** value.

(ONODEAVG): Originally used for scoring, this is a now a dead variable which serves no purpose.

Category OGAPARAM (13 members)

OGA_REC_RATE: The proportion of high-scoring OGA chromosomes that are permitted to recombine and contribute offspring to the next generation. **Floating-point** value between 0.0 and 1.0.

OGA_REC_REPL: The proportion of low-scoring OGA chromosomes that will be replaced by recombinants in the next generation. **Floating-point** value between 0.0 and 1.0.

OGA_MUT_RATE: The fraction of OGA Chromosomes whose parameters will be subjected to mutation after every training round. **Floating-point** value between 0.0 and 1.0.

OGA_MUT_AMT: The upper limit of the magnitude of mutations that are applied to genes. When a gene is mutated, its value will either be multiplied or divided by a value between 1.0 and OGA_MUT_AMT. **Floating-point** value between 1.0 (mutation has no effect) and infinity, but typically less than 2.0.

OGA_MUT_PROP: The proportion of genes to mutate within an OGA chromosome. **Floating-point** value between 0.0 and 1.0.

(OGA_KILL_PROP): The minimum proportion of genes that must be similar between two Chromosomes for these Chromosomes to be considered identical (resulting in the elimination of one or the other). **Floating-point** value between 0.0 and 1.0. *Note: The* *similarity of OGA Chromosomes is currently not checked, so changing this variable will have no effect on the analysis*.

(OGA_KILL_DIFF): The minimum proportional difference required between two genes for them to be considered ‘not identical’ for the purpose of identifying similar Chromosomes. **Floating-point** value between 0.0 and 1.0. *Note: The similarity of OGA Chromosomes is currently not checked, so changing this variable will have no effect on the analysis*.

GA_CHR: Number of OGA Chromosomes. **Integer** value > 0.

GA_EVO: Number of OGA Evolvables (separate Chromosome populations that do not interbreed). **Integer** value > 0.

GA_SEL: Number of OGA Selectables (separate Evolvable units that do not share members through migration). **Integer** value > 0.

*NUM_INPUTS: Number of inputs specified by each OGA chromosome. 0 < **Integer** value < Total number of indices. If IVO is false, then NUM_INPUTS will be equal to the number of indices in the input file.

OGA_TRAIN_ROUNDS: Number of OGA parameter/input optimization rounds to perform. **Integer** value > 0.

(MAX_GA_ROUNDS): The function of this variable has been rolled into NN_TRAIN_RUNS. **Integer** value > 0, but currently serves no useful purpose.

<u>Category STOPCOND (5 members)</u>

LR_TOOLOW (BP only): If learning rate decay is permitted, training stops if the learning rate drops below LR_TOOLOW. **Floating-point** value >= 0.0.

LR_CHECKROUND (BP only): The first BP training round where learning rate decay should be checked. **Integer** value > 0.

TINY_WEIGHT: The minimum permitted average weight of all links in the neural network. If the average drops below TINY_WEIGHT, training stops. **Floating-point** value >= 0.0.

CHECK_SCORE: To determine whether test score improvement has stalled, the current generalization score is compared to the score from CHECK_SCORE rounds ago. **Integer** value > 0.

NO_SCOREDIF: The minimum allowed difference between the current generalization score and the score from CHECK_SCORE rounds ago. If the difference is less than NO_SCOREDIF, then training stops. **Floating-point** value >= 0.0.

Category NNRANGESHARED (3 pairs of members)

NHIDNODEA, NHIDNODEB: The min/max number of neural network hidden nodes. **Integer** value > 0.

*NOUTNODEA, NOUTNODEB: The min/max number of neural network output nodes. This option will only be considered if the number of categories is 2, and the value specified is 1 or 2 output nodes. Otherwise, it will be equal to the number of different categories present in the index file. **Integer** value = 1 or 2.

NISBIASA, NISBIASB: If true, then the input and hidden layers will include a bias node. **Pseudo-boolean** value.

Category NNBPRANGE (7 pairs of members)

NLRNRATEA, NLRNRATEB: The min/max neural network learning rate. **Floating-point** value > 0.0.

NMOMENTA, NMOMENTB: The min/max neural network momentum term. **Floating-point** value >= 0.0.

NWEIGHTDECAYA, NWEIGHTDECAYB: The min/max proportional reduction of each connection weight for each training round. 0.0 <= **Floating-point** value <= 1.0.

NWTSTARTA, NWTSTARTB: The min/max round to begin weight decay. **Integer** value >= 0.

NLRNDECAYA, NLRNDECAYB: The min/max proportional reduction of the learning rate for each training round. 0.0 <= **Floating-point** value <= 1.0.

NLRNDECAYSTARTA, NLRNDECAYSTARTB: The min/max round to begin learning rate decay. **Integer** value >= 0.

NBATCHA, NBATCHB: Batch learning is used if this term is true. **Pseudo-boolean** value.

Category NNGARANGE (10 pairs of members)

NMUTRATEA, NMUTRATEB: The min/max fraction of IGA Chromosomes whose parameters will be subjected to mutation after every training round. **Floating-point** value between 0.0 and 1.0.

NMUTAMTA, NMUTAMTB: The upper limit of the magnitude of mutations that are applied to genes. When a gene is mutated, its value will either be multiplied or divided by a value between 1.0 and a given value (specified by the OGA Chromosome) between

NMUTAMTA and NMUTAMTB. **Floating-point** value between 1.0 (mutation has no effect) and infinity, but typically less than 2.0.

NMUTPROPA, NMUTPROPB: The min/max proportion of genes to mutate within an IGA chromosome. **Floating-point** value between 0.0 and 1.0.

NRECRATEA, NRECRATEB: The min/max proportion of high-scoring IGA chromosomes that are permitted to recombine and contribute offspring to the next generation. **Floating-point** value between 0.0 and 1.0.

NRECREPLA, NRECREPLB: The min/max proportion of low-scoring IGA chromosomes that will be replaced by recombinants in the next generation. **Floating-point** value between 0.0 and 1.0.

*NMIGRATEA, NMIGRATEB: The min/max rate of migration between IGA Evolvables. **Floating-point** value between 0.0 and 1.0. If there is only a single Evolvable, then migration will not occur.

NNUMCHRA, NNUMCHRB: The min/max number of IGA Chromosomes to create (the size of the population of GA nets). **Integer** value > 0.

NNUMEVOA, NNUMEVOB: The min/max number of IGA Evolvables to create (the number of non-interbreeding populations of GA nets). **Integer** value > 0.

NKILLFRACA, NKILLFRACB: The min/max range of the minimum proportion of genes that must be similar between two Chromosomes for these Chromosomes to be considered identical (resulting in the elimination of one or the other). **Floating-point** value between 0.0 and 1.0.

NKILLDIFFA, NKILLDIFFB: The min/max range of the minimum proportional difference required between two genes for them to be considered 'not identical' for the purpose of identifying similar Chromosomes. **Floating-point** value between 0.0 and 1.0.

CHANGE LOG

Version 2.0: Released May 2004

Major:

- The suite is now somewhat user-friendly and documented.
- Indices has been rewritten in C++: Execution takes seconds or minutes instead of hours.
- Support for GenBank files added to GetSeq.pl, parsing includes handling of pseudogenes. Also, different types of sequence (upstream, intergenic, etc.) can be explicitly requested.

- File format: A third column of information is required in GANN input files: the first three columns are now 'PosNeg', 'TrainTest', and 'SeqID'.
- Window numbering: Reversed, so that window '0' is always at the rightmost end of the sequence; i.e., closest to the start codon in upstream sequences.
- Can now load a trained neural network and run it on an index file without having to train up a whole OGA.
- Reorganized 'Indices' such that mapping rules (for Twist, Roll, etc.) are no longer hard-coded, but are user-specified in a file. Also added support for counting specific oligonucleotides, and explicit (but general) functions for counting nucleotide intervals and complete sets of nondegenerate n-mers. Also integrated the generation of negative control (shuffled) sets within 'Indices'.
- Added support for > 2 sets within the 'PosNeg' category: this required more flexible data structures and new error functions for the MLP classes.

Minor:

- Combine can now read its own output files as input.
- Combine now lets you specify a filehandle to narrow the list of files under consideration.
- Combine also explicitly allows the choice to include or exclude global variables, Z-scores, and a more limited window range.
- Exorcized some dead code from the GANN core.
- Added an option to create neural networks with bias nodes.

Version 1.0: Unreleased

- Object-oriented implementation of multi-layer perceptrons (class MLP), with derived classes BPNet (backprop) and GANet (genetic algorithm)
- Global variable values are 'const' and compiled with the GANN program: Fast, but requires a recompile every time these values are changed.
- Window numbering: The windows are numbered in increasing order from left to right within the sequence; for upstream sequences this means that the highest window number is proximal to the start codon.
- Wrote a script (collectScores.pl) to collect and summarize the results of GANN runs.
- Changed the squashing function from sigmoidal to tanh, lookup table generated to speed up execution.
- Started the migration from explicit 'new' and 'delete' of pointers to C++ STL classes in 'GANN'

TO DO

The rate of completion of these items will depend in large part on the usage of this software by others, and the feedback I get from users. More ambitious upgrades may appear in this list (or in the software, for that matter) without warning.

- Implement more positive and negative control techniques within Combine, or as separate programs / scripts.
- Test GetSeq on more genomes.
- Add support for more GenBank tags within 'GetSeq.pl', in particular for the 'join' tags.
- Ensure that all input files to 'combine' have the same entries in the same order to avoid bizarre combinations of sequence indices. Also, modify ‘combine’ so it deals with poorly formatted files in a more graceful manner.
- Once a neural network has been trained on a given subset of indices, make it easier to generate that same subset of indices for a new set of sequences.
- New class within 'Indices': Oligonucleotide counts, but with some degree of mismatch permitted. Also, reintroduce nucleotide counts that receive bonuses when adjacent nucleotides are of the same type.
- Object-oriented implementation of search types within 'Indices': might make it easier to add new types and maintain the existing ones.
- Implement a better string-searching algorithm in 'Indices': Instead of storing strings in the sequence to value maps, it would be better to store precomputed bit masks and use them for searching later on.
- Add exceptions and error reporting to GANN core.