ENZO

Evolution of Neural Networks

by

Heinrich Braun and Thomas Ragg

University of Karlsruhe

Institute for Logic, Complexity and Deduction Systems

User Manual and Implementation Guide, Version 1.0

Contents

Part I

Enzo User Manual

1 General Introduction

Summary

The construction of a neural network to a given problem specification is a difficult optimization problem: Which topology (number of layers, number of units per layer, connectivity of units) and which values for the network coefficients (weights, threshold) gains the optimal performance? Our evolutionary network optimizing system (Enzo) uses the paradigm of evolution for optimizing the topology and the paradigm of learning for optimizing the coefficients. Particularly, Enzo evolves a population of networks by generating offsprings thru mutating the topology of the parent network and by optimizing the coefficients with our fast gradient descent algorithm RPROP. For measuring the performance (resp. the fitness) we can use different criteria: learning error (error on learning set), generalization capability (error on the test set), hardware complexity (number of units and weights), runtime (number of layers), etc. Using several heuristics for speeding up the training time of the offsprings Enzo can efficiently optimize even large networks with 5 000 weights and 50 000 training patterns.

1.1 Introduction

The basic principles of evolution as a search heuristic may be summarized as follows. The search points or candidate solutions are interpreted as individuals. Since there is a population of individuals, evolution is a multi-point search. The optimization criterion has to be one-dimensional and is called the fitness of the individual. Constraints can be embedded in the fitness function as additional penalty terms. New candidate solutions, called offsprings, are created using current members of the population, called parents. The two most used operators are mutation and recombination (crossover). Mutation means that the offspring has the same properties as its single parent but small variations. Whereas recombination (crossover) means that the offspring's properties are mixed from two parents. The selection of the parents is randomly but biased, preferring the fitter ones, i.e. fitter individuals produce more offsprings. For each new inserted offspring another population member has to be removed in order to maintain a constant population size. This selection can be done randomly or according the fitness of each member. Particularly, that means in the first case that the expected lifetime is equal for all members, whereas in the second case the fitter will live longer, i.e. the fittest may even survive for ever.

The design of neural networks incorporates two optimization problems. First of all the topology, i.e. number of hidden units and their interconnection structure, and second the tuning of the net work parameters i.e. weights. Therefore, we have to solve a mixed optimization problem, discrete for the topology and continous for the network parameters. The standard approach is to use the intuition of the designer for defining the topology and to use a learning algorithms (e.g. gradient descent) for adjusting the free parameter.

The published results for using evolutionary algorithms for the parameter optimization instead of gradient descent like backpropagation suggest, that this is only efficient, when gradient descent is not possible (e.g. activation function of neurons is not differentiable or the interconnection structure is not feed forward but contains cycles) or unsuccessful for sparse topologies, since gradient descent is much faster.

On the other hand, optimization of the topology incorporates optimization of the parameters, since evaluating the fitness of a topology means evaluation of the network behavior for which we need an optimal instantiation of the according network parameters (training). Meanwhile, there are some published approaches which use evolution for the discrete topology optimization and use gradient descent just for the fitness evaluation of the topology. Since training rsp. gradient descent of a neural network is even for middle sized networks a time consuming task, these investigations are limited to small networks. In the following we describe an hybrid approach combining evolution and gradient descent such that even large networks with over 5000 conne ctions and training sets with over 50 000 patterns (ca. 3 Mbyte) can be efficiently handled.

1.2 ENZO , - Our Evolutionary Approach

Every heuristic for searching the global optimum of difficult optimization problems has to handle the dilemma between exploration and exploitation. Priorizing the exploitation (as hill-climbing strategies do) bears the danger of getting stuck in a poor local optimum. On the other hand, full explorative search which guarantees to find the global optimum, uses vast computing power. Evolutionary algorithms avoid getting stuck in a local optimum by parallelizing the search using a population of search points (individuals) and by stochastic search steps, i.e. stochastic selection of the parents and stochastic generation of the offsprings (mutation, crossover). On the other hand, this explorative search is biased towards exploitation by biasing the selection of the parents preferring the fitter ones.

This approach has proven to be a very efficient tool for solving many difficult combinatorial optimization problems (Goldberg, 1989, Reeves, 1993, Schwefel, 1995). A big advantage of this approach is its general applicability. There are only two problem dependent issues: The representation of the candidate solutions as a string (genstring = chromosome) and the computation of the fitness. Even though the choice of an adequate representation seems to be crucial for the efficiency of the evolutionary algorithm, it is obvious that in principle both conditions are fulfilled for every computable optimization problem.

On the other hand, this problem independence neglects problem dependent knowledge as e.g. gradient information. Therefore the pure use of evolutionary algorithms may have only modest results in comparison to other heuristics, which can exploit the additional information. For the problem of optimizing feedforward neural networks we can easily compute the gradient by backpropagation. Using a gradient descent algorithm we can tremendously diminish the search space by restricting the search to the set of local optima.

This hybrid approach uses two time scales. Each coarse step of the evolutionary algorithm is intertwined with a period of fine steps for the local optimization of the offspring. For this approach there seems to be biological evidence, since at least for higher animals nature uses the very same strategy: Before evaluating the fitness for mating, the offsprings undergo a longer period of fine tuning called learning. Since the evolutionary algorithm uses the fine tuning heuristic as a subtask, we can call it a meta-heuristic. Obviously, this meta-heuristic is at least as successful as the underlying fine tuning heuristic, because the offsprings are optimized by that. Our experimental investigations will show, that the results of this meta-heuristic are not only as good but impressively superior to the underlying heuristic.

In the natural paradigm the genotype is an algorithmic description for developing the phenotype, which seems not to be an invertible process, i.e. it is not possible to use the improvements stemming from learning (fine tuning) for improving the genotype as Lamarck erroneously believed. In our application however, there is no difference between genotype and phenotype, because the matrix of weights, which determines the neural network, can be linearly noted and interpreted as a chromosome (genstring). In this case Lamarck's idea is fruitful, because the whole knowledge gained by learning in the fine tuning period can be transferred to the offsprings (Lamarckism).

The strengths of our approach stem mainly from this effect in two ways: Firstly, since the topology of the offsprings is very similar to the topology of the parents transferring the weights from the parents to the offsprings diminishes impressively the learning time by 1-2 orders of magnitude (in comparison to learning from the scratch with random starting weights). This also implies, that we can generate 1-2 orders of magnitude more offsprings in the same computation time. Secondly, the average fitness of these offsprings is much higher: the fitness distribution for the training of a network topology with random initial weights will be more or less Gaussian distributed with a modest average fitness value whereas starting near the parental weights (remind the topology of the offspring is similar but not the same as that of its parents) will result in a network with a fitness near the parental fitness (may be worse or better). That means, whenever the parental fitness is well above the average fitness (respectively its topology) then the same may be expected for its offspring (in case using the parental weights). Moreover, our experiments have shown for the highly evolved 'sparse' topologies that with random starting weights the gradient descent heuristic did not find an acceptable local optimum (solving the learning task), but only by inheriting the parental knowledge and initializing the weights near the parental weights.

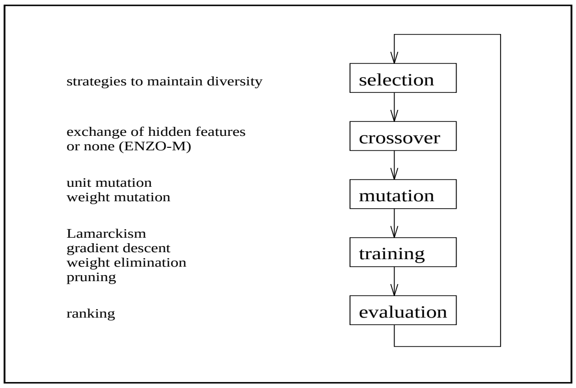


Figure 1: Evolution cycle of ENZO

Summarizing, our algorithm briefly works as follows (cf. fig. 1). Taking into account the user's specifications ENZO generates a population of different networks with a given connection density. Then the evolution cycle starts by selecting a parent, preferring the ones with a high fitness ranking in the current population and by generating an offspring as a mutated duplication of this parent. If crossover is chosen, one or few hidden features (=hidden units) of a second parent may be randomly added. Each offspring is trained by the best available efficient gradient descent heuristic (RPROP, see (Riedmiller & Braun, 1993)) using weight decay methods for better generalization. By removing negligible weights, trained offsprings may be pruned and then re-trained. Being evaluated an offspring is inserted into the sorted population according to its determined fitness value, thereby removing the last population element. Fitness values may incorporate any design criterion considered important for the given problem domain.

1.3 Mutation

Besides of the widely used link mutation we also realized unit mutation which is well suited to significantly change the network topology, allowing the evolutionary algorithm to explore the search space faster and more exhaustive. Our experiments show that unit mutation efficiently and reliably directs the search towards small architectures.

Within link mutation every connection can be changed by chance. The probability for deleting a link should be correlated with the probability, that this deletion doesn't decrease the performance significantly. For that, we prefer links with small weights for deletion, whereas the probability of adding is equal for all links:

Weighted Link mutation: For each link the probability to be deleted is $p_{del} * N_\sigma(0, w)$ where p_{del} and σ are set by the user, $N_\sigma(0, w)$ means a normal distributed value with mean 0 and variance σ und w denotes the absolute weight value of the considered link. The probability for adding a link is p_{add} constantly.

We may call this soft pruning, since not only the weights below the threshold σ are pruned, but very small weights ($|w| << \sigma$) with probability about p_{del}, big weights ($|w| >> \sigma$) with probability about 0 and a soft interpolation in between. By the factor $N_\sigma(0, w)$ we intent to approximate the probability that the deletion of a link effects no significant deterioration of the performance. If we choose p_{del} such that only a few links are deleted by each weight mutation we get the following heuristic:

Soft pruning: Test a few weights for pruning preferring small weights. Whenever this pruning effects no significant deterioration, this variant will survive and will be subject to more pruning, - else this offspring is classified as failure and therefore not inserted in the population.

Therefore it is not necessary to estimate the effects of the deletion of a weight as is done by other heuristics (e.g. optimal brain damage, optimal brain surgeon)—just try and test it.

In contrast to link mutation, unit mutation has a strong impact on the network's performance. To improve our evolutionary algorithm we developed two heuristics which support unit mutation: the prefer-weak-units (PWU) strategy and the bypass algorithm. The idea behind the PWU-heuristic is to rank all hidden units of a network according to their relative connectivity ($\frac{act.connections}{max.connections}$) and to delete sparsely connected units with a higher probability than strongly connected ones. This strategy successfully complements other optimization techniques, like soft pruning, implemented in ENZO .

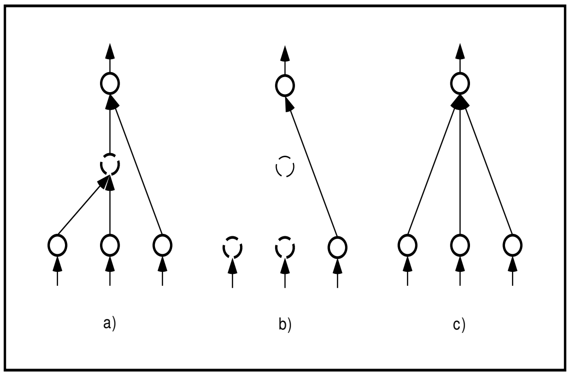


Figure 2: Bypass algorithm: a) original network b) after deletion of the middle unit c) with added bypass connections

The bypass algorithm is the second heuristic we realized. Other than adding a unit, deletion of a unit can result in a network which is not able to learn the given training patterns. This can happen because there are too few hidden units to store the information available in the training set. But even if there are enough hidden units, deleting a unit can destroy important data paths within the network (fig. 2a and b). For that reason we restore deleted data paths before training by inserting bypass connections (fig. 2c). By that, the nonlinear function computed by the subpart of the neural network, which was connected to the deleted unit formerly, is now approximated by a linear function using shortcuts (bypass connections). The application of the bypass algorithm significantly increases the proportion of successful generated by unit mutation. Both the number of networks with devastated topologies decreases and the generated nets need less epochs to learn the training set, because they are not forced to simulate missing bypass connections by readjusting the weights of the remaining hidden units.

1.4 Benchmarks

Some benchmark problems are distributed with ENZO , three simple benchmarks with only a few minutes computing time necessary (TC-Problem,Encoder,XOR) and two larger benchmarks (Spirals,Recogintion of digits). Some benchmarks are also described in the following. Before using ENZO for larger problems, it is worth investigating some time in parameter tuning of benchmark problems to get an impression on the influence of single parameters and the dependencies between parameters.

1.4.1 TC problem

The task is to correctly classify Ts and Cs given a $4x4$ pixel input matrix (figure 3, see also (McDonell & Waagen, 1993)). The pattern set contains all possible Ts and Cs, that is they can be translated and rotated. In total there are 17 Ts and 23 Cs. A straightforward network uses the pixel representation as input units, has some hidden units and one output unit, that classifies Ts with 1 and Cs with 0. The topology of the network is than $16-16-1$ with full connection, i.e., 288 weights. Obviously the input layer contains redundant information.

The task for the genetic algorithm is to eliminate redundant input units and furthermore the topology of the network, without any loss of classification performance.

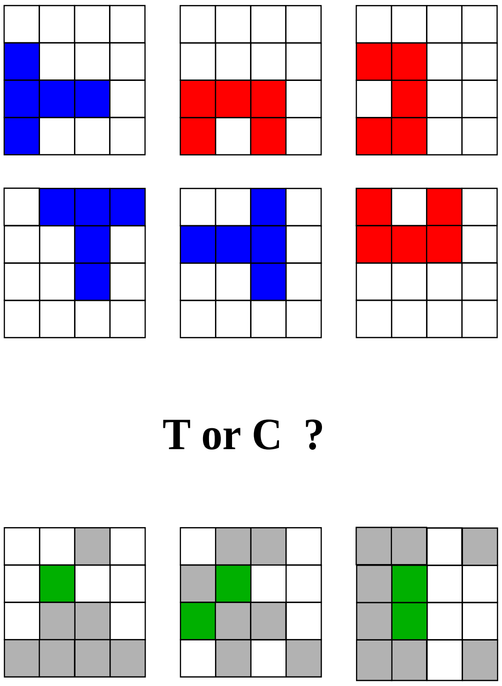


Figure 3: The figure shows Ts and Cs represented with a 4x4 pixel matrix. In the bottom row, several input units were cut off (gray color). The network is still able to distinguish every T from every C. Can you ?

The neural network that originally had nearly 300 links was already impressively reduced to a 10-2-1 net with 27 links left by ENZO and in a second approach to 8-2-1 with only 18 links.

Reference	topology	#links
Original net	16-16-1	288
McDonnell	15-7-1	60
ENZO 94	10-2-1	27
ENZO 95	8-2-1	18

Table 1: *TC-Problem. Note the decreased number of input units due to input-unit-mutation.*

1.4.2 Nine Men's Morris

With ENZO we investigated networks learning a control strategy for the endgame of **Nine Men's Morris**. The table 2 shows the performance of three nets: the first network was the best hand-crafted network developed just by using backpropagation (SOKRATES,(Braun *et al.*,)), a second network was generated by ENZO (**?**) and a third network we got by ENZO additionally rating the network size in the fitness function (**?**). Networks optimized by ENZO show a significantly better performance than the original Sokrates net. Further, that superior performance is achieved with smaller nets. ENZO was able to minimize the network to a 14-2-2-1 architecture deleting not only hidden units but also the majority of the input neurons.

System	topology	#weights	performormance
Sokrates	120-60-20-2-1	4222	0.826
ENZO -1	120-60-20-2-1	2533	1.218
ENZO -2	**14-2-2-1**	**24**	1.040

Table 2: *Socrates was the best handcrafted network, the fitness criteria for* ENZO *-1 was performance and for* ENZO *-2 additionally network size.*

1.4.3 Thyroid gland

Thyroid gland diagnostic is a real-world benchmark we used to test our algorithm [**??**]. This task requires a very good classification, because 92% of the patterns belong to one class. So a useful network must classify much better than 92%. A further challenge is the large number of training patterns (nearly 3800) which exceeds the size of toy problem's training sets by far. The evolved network had the same performance as in [**??**], but 4 input units less and only 20% of the weights.

	topology	#weights	performance
Schiffmann	21-10-3	303	98.4%
ENZO	17-1-3	66	98.4%

Table 3: *Thyroid gland diagnostic, - decreasing network size and removing redundant input units without deteriorating performance*

1.4.4 Classification of handwritten digits

Classification of handwritten digits was the largest problem we tackled with ENZO (**?**). We compared the classification performance of our evolved neural networks with that of a commercially used polynomial classifier of degree two. Trained on the same 50,000 pattern subset of the NIST data base using the same features as the neural net the polynomial classifier achieves a classification rate of 99.06% correct. The results in table 4 shows that both classification approaches perform equally well. The major difference is in the number of free parameters: while a 2^{nd} degree polynomial classifier uses 8.610 coefficients our nets range from less than 1,600 to 3,300 links. As a consequence the classification time in practical application is reduced to 20% of time needed by a PC. Another advantage is the possibility to obtain a specialized net for a given time-accuracy tradeoff. By means of the fitness-function the user can support the evolution of either nets with few links, risking a small drop in performance, or more powerful nets with some links more.

links	correct	#weights : misclassified
892	98.05	1 : 2
1,648	98.71	1 : 5
3,295	99.16	1 : 1500

Table 4: *Classification of handwritten digits, - evolved networks for different ratings of network size (#weights) versus performance (miscl.) in the fitness function*

1.5 Conclusion

ENZO combines two successful search techniques: gradient descent for an efficient local weight optimization and evolution for a global topology optimization. By that it takes full advantage of the efficiently computable gradient information without being trapped by local minima. Through the knowledge transfer by inheriting the parental weights both learning is speeded up by 1-2 orders of magnitude and the expected fitness of the offspring is far above the average for its topology. Moreover, ENZO impressively thins out the topology by the cooperation of the discrete mutation operator and the continuous weight decay method. For this the knowledge transfer is again crucial, because the evolved topologies are mostly too sparse to be trained with random initial weights. Additionally, ENZO tries also to cut off the connections to eventually redundant input units: For the Nine Men's Morris problem ENZO found a network with better performance but only 12% of the input units originally used. Therefore ENZO not only supports the user in the network design but also determines the salient input components.

2 Who should use ENZO

2.1 History and purpose of ENZO

ENZO was designed to optimize the topology of neural networks as well as their performance. So far this version supports the optimization of multilayer perceptrons. Elman networks, TDNNs and RBF networks are currently under investigation.

This version of ENZO uses the Stuttgarter Neural Network Simulator (SNNSv4.1, kernel and function library) for manipulating neural networks. All simulators can be supported as long as they offer a functional interface to manipulate networks.

ENZO should be a powerfool tool for everybody who uses neural networks and who is interested in faster, smaller and better networks. It is not necessary to have any knowledge about genetic algorithms, but it makes the system easier to comprehend. See (Goldberg, 1989, Reeves, 1993, Schwefel, 1995) for an introduction.

The flexible design of ENZO provides a tool that is usable for many tasks, when dealing with neural networks. That is, instead of optimizing topologies, one can use it as well as a batch program to train several networks just by changing the command file in an appropriate way. Adding your own modules allows you to tailor the program to your desire.

2.2 Where to get ENZO

ENZO is available via anonymous ftp at the same site as the SNNS simulator. The host is

`ftp.informatik.uni-stuttgart.de (129.69.211.2)`

in the directory

`/pub/SNNS`.

Check there for further information (Readme.ENZO) and the file ENZO.tar.Z or ENZO.tar.gz! Before extracting the tar-files note that there is no installation script by now. You should have no problems if ENZO (resp. the tar-files) are located on the same directory level as SNNSv4.1. Uncompress the tar-file and extract ENZO with

```
tar -xvf ENZO.tar
```

in the current directory.

The directory ENZO contains a makefile to compile the program. The subdirectory ENZO/src contains all sources. The subdirectory ENZO/benchmarks contains some benchmarks. See also section **??**. The subdirectory ENZO/doc contains the documentation (with LaTeX sources).

2.3 Mailing list

There exists a mailing list for ENZO . If you want to be added to the list, send a message to

`enzo-request@ira.uka.de`.

Post your messages, questions, comments etc. to

`enzo@ira.uka.de`.

3 Design and Interface of ENZO

The design of ENZO provides a great flexibility. The specialized knowledge of how to perform a certain evolution step is located in the modules (right lower corner of the ENZO block in figure 4. They are combinable like toy blocks and easily extensible. A population manager takes care of handling the individuals as well as the pattern sets. The neural network simulator is hidden behind a functional interface. ENZO also offers the possibility to use the network description language CuPit . If one is familiar with CuPit and interested in using it with ENZO please send an email to `enzo-request@ira.uka.de`. For more information about CuPit see `http://wwwipd.ira.uka.de/~hopp/cupit.html`.

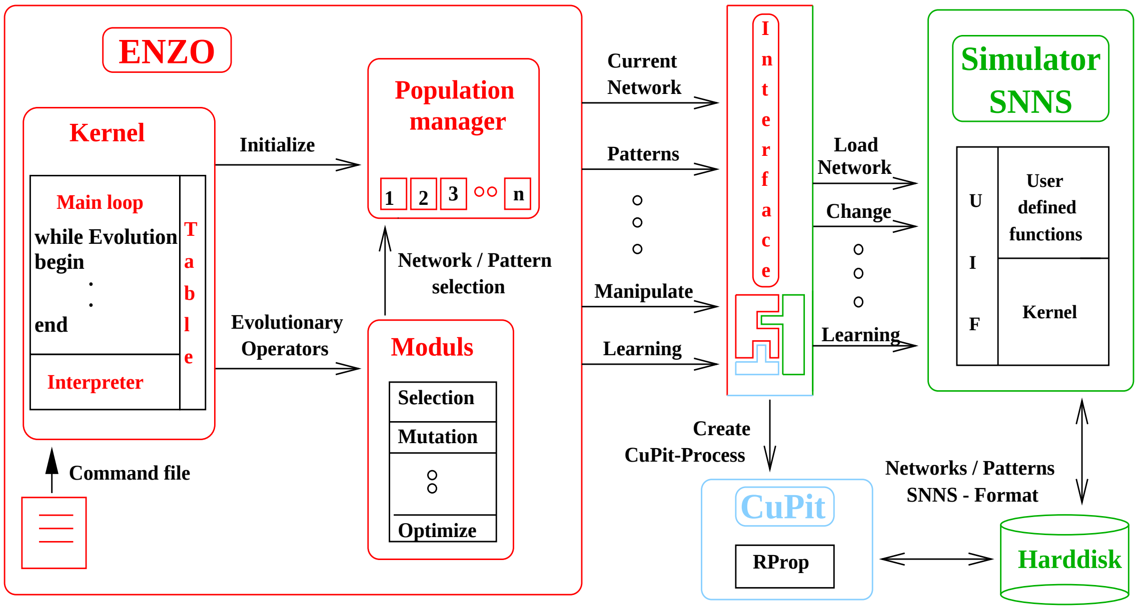


Figure 4: The figure shows the main parts of ENZO and the interface to the neural network simulator, e.g., the SNNS simulator. The Interface contains about 100 functions to perform several network operations.

4 Installing and running ENZO

4.1 Installation

Unfortunately, for this first published version of ENZO no installation script exists. You don't have to change any makefiles as long as ENZO is located on the same directory level as the SNNS simulator. (Expected name is SNNSv4.1). If this is not the case, use symbolic links or adapt the makefiles.

To install ENZO do the following:

1. Make sure ENZO is at the same directory level as SNNSv4.1
2. Type `cd ENZO` and than `make`. That should compile all libraries as well as the executable `enzo`. The executable is located in the directory `ENZO`
3. If you want to use the X-history window, type `make xgraf`. You may need to adapt the library and include path in the makefile in `ENZO/src/history/Xgraf`.

4.2 Running ENZO

ENZO is run as a background (UNIX-) process. For small problems, a simple X-Window visualization of the fitness function is usable. The networks can be analyzed using the graphical user interface of SNNS. In near future, some tools will be provided with each standard history module, to visualize the results.

To run ENZO one simply types:

`enzo` *cmd_file [logfile [seed]]*

If no *log file* is given, the output is written to *stderr*.

```
Pre-evolution();              // Initialize, ...
repeat
    Selection();              // Select individuals for reproduction
    Crossover();              // Cross two individuals
    Mutation();               // Mutate an individual
    Optimization();           // Locally optimize (train) all offsprings
    Evaluation();             // Compute fitness of all offsprings
    History();                // Save protocol information
    Survival();               // Merge offsprings into population
until   Stop-evolution();     // Check stopping criteria

Post-evolution();             // Save reults, clean up, ..
```

Figure 5: The figure shows the main loop of ENZO . The evolutionary operators are called in the shown sequence.

ENZO starts by reading the *command file*. A sample command file is given in chapter A. Via the command file modules can be activated through a key word and their parameters can be set. The genetic operators are called sequentially as shown in figure 5. Each operator can consist of several modules (or be empty). The modules are combined by specifying their key words in the command file. They are called in the sequence of the appearance of the key words. Note that one module can appear several times in this sequence. Figure 6 illustrates the relationship between modules and operators.

4.3 The command file

All possible key words are defined by the modules. For details see the description of the modules in chapter 5. A dispatcher passes the key words and possible parameters to all modules. Each module picks the information it is interested in and performs necessary actions. The sequence of key words is only important in the way that the functionality of the resulting operator depends on the order of the keywords, e.g., the optimization operator in figure 6 has another functionality if prune would be called before learnSNNS.

Still it is good style to keep certain entries in different parts:

1. **Files:** You should specify the file names of the networks and patterns in this part. Also the prefix for output files should be given. This has the advantage that one sees immediately, what task is optimized and which files are involved.

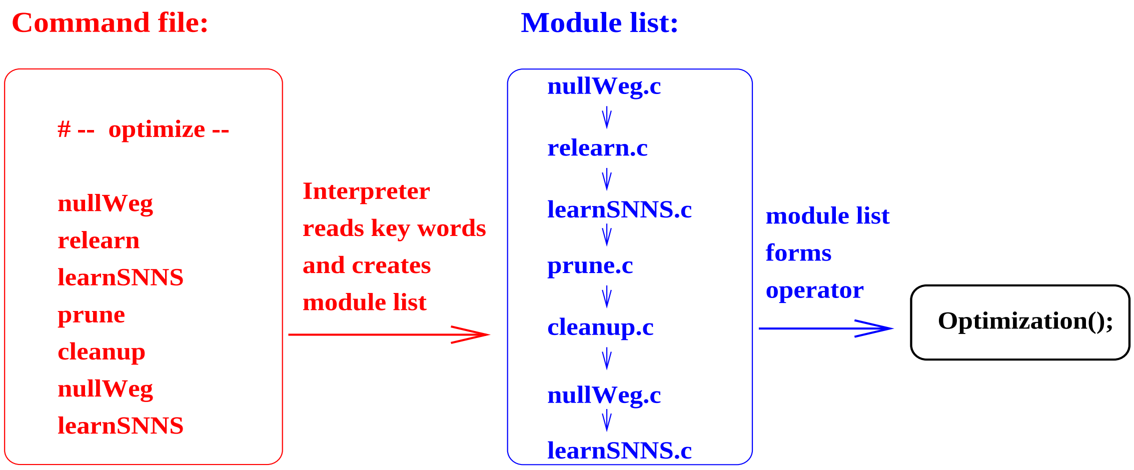


Figure 6: The figure shows the relationship between modules and operators. The user can specify the key words of the modules in the command file, which will activate the modules,e.g., the interpreter adds them to a module list which forms the evolutionary operator.

2. **Modules:** You should specify which modules form the evolutionary operators. Every module defines a key word for its activation. The key words should be in the typical order, e.g., pre-evolution before selection before crossover etc. This part says which modules are to use.

3. **Parameters:** All parameters of all used modules should be set here. The order of key words should be the same as for modules. If a parameter is set several times the last appearance is used. This part decides how the evolution is done in detail.

A sample command file is shown in chapter A.

5 Module description

The following sections describe the modules which are currently available for ENZO . Each section corresponds to an operator, each subsection corresponds to a module. Firstly the key word of the module is given, followed by the description of its parameters. Optional parameters are given in brackets. All modules have sensible default values for their parameters. Some notes on important parameters can be found in chapter 6. Each section is closed by a functional description of the module and a sketch of the algorithm, if necessary.

5.1 Pre-evolution

5.1.1 Create an initial population

key word: **initPop**

`gensize` [x]
This parameter sets the maximal number of networks in the parent population.
Default: `POP_SIZE_VALUE` (30)

`popsize` [x]
This parameter sets the number of offsprings to create each generation.
Default: `OFF_SIZE_VALUE` (10)

`network` [x]
This string contains the filename of the reference net. Each created net in the population gets the same topology structure as the reference net.
Default: enzo.net

`initFct` [x]
This string contains the name of the SNNS init-function. The starting values of the weights and biases will be set by this function.
Default: Randomize_Weights

`initParam` [x]
These 5 parameters contains the parameters for the init-function. For the meaning of these parameters please see the SNNS manual.
Default: -1.0 1.0 0.0 0.0 0.0

The module *initPop* loads the reference net (via SNNS) and copies this net to all members of the parent population. After that all networks of the population are initialized with the SNNS initial function.

Algorithm *initPop*:

Load the reference net;
Set the names of all units in the reference net;
forall (members of the starting population) **do**
    Copy the reference net to the new net;
    Initialize the net with *initFct* and *initParam*;
    Set the *initFct* of the net to *ENZO_noinit*;

5.1.2 Load a starting population

key word: **loadPop**

`network` [x]
This string contains the prefix of the filename, where the networks are stored in.
Default: `enzo`

`popsize` [x]
This parameter x sets the number of networks for the starting population.
Default: `POP_SIZE_VALUE` (30)

The module *loadPop* loads networks for the starting population. The filename for the reference net consists of the prefix *network* and the extension `_ref.net`. The filename for the other networks consists of the prefix *network* and an extension containing a number, e.g., `network_0000.net`. It is important that all hidden units of the reference net and the other networks of the starting population get the same unit name in SNNS. The purpose of this module was to restart an evolution from a stopped process. The postfix of network names are just in the way Enzo stored them.

5.1.3 Creating a population using the nepomuk library

key word: **genpopNepo**

`popsize` [x]
This parameter sets the maximal number of networks in the parent population.
Default: `POP_SIZE_VALUE` (30)

`gensize` [x]
This parameter sets the number of offsprings to create each generation.
Default: `OFF_SIZE_VALUE` (10)

The module creates a population of networks using the nepomuk library. Three parts of the population are distinguished: the reference net, the parent population and the offspring population. Note that only memory for *popsize* + *gensize* + 1 networks is allocated, but no networks are created at that time.

5.1.4 Load standard SNNS pattern sets

key word: **loadSNNSPat**

`learnpattern` [x]
This string contains the filename for the learning patterns.
Default: -

`testpattern` [x]
This string contains the filename for the test patterns. The test patterns are used to determine the fitness of the networks for the genetic algorithm.
Default: -

`crosspattern` [x]
This string contains the filename for the validation patterns. The validation patterns are used to determine the efficiency of networks. These patterns should not be used in the learning phase or to determine the fitness of the networks.
Default: -

The module *loadSNNSPat* loads the three pattern sets with the original SNNS-function. The filename must contain the extension `.pat` for the SNNS pattern files. The number of pattern sets to be managed is restricted to 3. To use more sets you have to increase the maximum number defined in the population manager.

5.1.5 Learning during the pre-evolution

key word: **initTrain**

`initLearnfct` [x]
: This string contains the name of the SNNS learning function.
Default: Rprop

`initLearnparam` [x]
: These 5 parameters contains the values of the parameters for the learning function. For further informations see the SNNS manual. Default: 0.0 0.0 0.0 0.0 0.0

`initMaxepochs` [x]
: This parameter x contains the maximum number of periods for the learning algorithm. After this maximum the module *initTrain* will automatically stop the learning function.
Default: 50

`initMaxtss` [x]
: This parameter indicates the maximal tolerable learning error. The error is normalized by the number of learning patterns and the number of output units. The module *initTrain* will terminate the learning function if the learning error is less than this threshold.
Default: 0.5

`initShuffle` [x]
: This switch indicates whether the sequence of the learning patterns is changed after each learning period or not. If the switch is turned on, then the module *initTrain* will use the SNNS function *shuffle*
Default: yes

The module *initTrain* is an alternative version to the standard learning module *learnSNNS*. It is possible to use different learning functions and parameters in the pre-evolution phase than in the optimization phase. In the sense of lamarckism, where offsprings get the strength of their weights directly from their parents, it is useful to have this opportunity. This leads to less learning epochs for offsprings in comparison to networks of the starting population, i.e., randomly initialized networks.

Algorithm *initTrain*:

for (each net) **do**
    Set the actual SNNS-learning function *initLearnFct*;
    while (Epochs < *initMaxEpochs*) **and** (tss > *initMaxTss*) **do**
        Learn one epoch with *initLearnParam*;

5.1.6 Random selection of input units

key word: **inputInit**

`minNoInput` [x]
This parameter sets the lower bound of the number of active input units.
Default: 1

`maxNoInput` [x]
This parameter sets the upper bound of the number of active input units.
Default: *Number of all input units*

The module *inputInit* randomly selects between *minNoInput* and *maxNoInput* input units for the net of the starting generation. The other input units deactivated in the network structure by deleting all incoming and outgoing weights of this units. The purpose of this module is to increase the diversity of the starting population by creating different input layer topologies.

5.1.7 Look for the optimal number of hidden units.

key word: **optInitPop**

`maxtss` [x]
This parameter sets the value for the stop criterion of the learning module. In this module it is used to decide if a net can learn the patterns or not.
Default: 0.5

`learnModul` [x]
This string sets the name of the module, which contains the learning function during the optimization phase.
Default: `learnSNNS`

The module *optInitPop* tries to find the minimum number of hidden units, which are required to learn the given problem properly. This number of hidden units is computed with a binary search. When this minimum topology is found, the rest of the networks will be randomly created with a number of hidden units between the found number and the maximum number of hidden units. The purpose of this module is to reduce the network size of the parent networks to a sensible size to speed up the evolution process.

Algorithm *optInitPop*:

```
Load the reference net;
Name the hidden units in the reference net;
lowerBound = 0;
upperBound = # hidden units;
learned= 0;
repeat
   learned++;
   hiddenUnits = (lowerBound + upperBound) div 2;
   copy the reference net;
   delete (#hidden units in reference net - hiddenUnits) from the offspring;
   train the offspring with learnModul;
   if (offspring is trained well) then
      upperBound = hiddenUnits;
   else
      lowerBound = hiddenUnits + 1;
until (upperBound ≤ lowerBound) or (learned = popsize)

for (i = learned + 1 to popsize) do
   copy the reference net;
   hiddenUnits = Rand(lowerBound, #hidden units);
   delete (#hidden units - hiddenUnits) from the offspring;
```

5.1.8 Create a population of networks from one special network

key word: **startPop**

`popsize` [x]
: This parameter sets the number of elements in the population.
Default: `POP_SIZE_VALUE` (30)

`network` [x]
: This string contains the filename of the reference network.
Default:

`startnet` [x]
: This string contains the filename of the master network, which structure is copied to all the other networks.
Default:

`initFct` [x]
: This string contains the name of the SNNS init-function. This function is used to initialized all created networks.
Default: `ENZO_noinit`

`initParam` [x]
This five parameters contains the values for the function parameters of the SNNS init-function *initFct*. For the meaning of the parameters please see the SNNS manual.
Default: -1.0 1.0 1.0 0.0 0.0

The module *startPop* loads the reference network and a special master network. The topology of the master network is copied to all the other networks in the start-population. Afterwards all weights and biases of the networks will be initialized with the *initFct*. The idea is to take a good network to initialize the genetic search. Another possibility is to overcome the limitation of the maximal topology by using a much bigger reference network than master network, with the master network maybe untrained. If the master network is already locally optimized, one should use the initialization function *ENZO_noinit*.

Algorithm *startPop*:

Load the reference and the master net;
forall (Elements in the start-population) **do**
  Copy the master network to the network;
  Initialize the networks randomly with *initFct* and *initParam*;
  Set *initFct* to *ENZO_noinit*;

5.1.9 Random selection of weights

key word: **weightInit**

`weightProb` [x]
This parameter sets the probability p_{exists}. Each weight will be deleted probability $1 - p_{exists}$. Default: 1.0

The module *weightInit* deletes weights from a net of the start-population randomly. This could by necessary to increase the diversity of the population or to reduce the free dimensions of the network.

Algorithm *weightInit*:

forall (Weights in the net) **do**
  if (Rand(0,1) $> p_{exists}$) **then**
    delete the weight in the net;

5.2 Stopping condition

5.2.1 Normal stopping

key word: **stopIt**

`maxGenerations` [*cnt*]
: Stops the evolution after *cnt* generations; Is *cnt* not specified, after the first generation.

5.2.2 Stopping by error

key word: **stopErr**

`no parameter`

Stops the evolution, if parents or offsprings are not valid networks. This usually happens, if the initialization is missing.

5.3 Selection

5.3.1 Uniform selection

key word: **unifSel**

`NoOfOffsprings` k
: Sets the number of new offspring each generation to k.

`selProb` p_{sel}
: Sets the probability of selection to p_{sel}.

The selection probability is `prob`.

5.3.2 Selection of parents preferring the better networks

key word: **preferSel**

`gensize` [*gen*]
: This parameter sets the number of networks in the offspring population.
Default: 10

`preferfactor` [x]
: This parameter x sets the bias for selecting fitter networks. A value $x \geq 1$ means that the better networks will be preferred, a value $0 \leq x \leq 1$ means that the poor networks. A value $x = 2$ means that the first quarter of the population is as often selected as the rest of the population. A value $x = 3$ means that the first 12.5% of the population is as often selected as the rest of the population.

Default: 3.0

This module selects the given number of parents for the reproduction process (mutation and crossover). Better networks are selected with a higher probability.

Algorithm *preferSel*:

n = number of networks in the parent population;
p = preferfactor;
$parentNo = (rand(0,1)^p) * n$
Select the network with number *parentNo*;

5.4 Mutation

5.4.1 A simple weight mutation

key word: **simpleMut**

`probadd` $[p_{add}]$
This parameter x indicates the probability p_{add} for inserting a non existing weight.
Default: 0.0

`probdel` $[p_{del}]$
This parameter x indicates the probability p_{del} for deleting an existing weight.
Default: 0.0

`initRange` $[f]$
The parameter x determines the interval [-range, range] where inserted weights are randomly selected from.
Default: 0.5;

The module *simpleMut* executes a simple kind of weight mutation. Each existing weights of the offspring will be deleted with the probability p_{del}, each weight, that exists in the reference net and not in the offspring will be inserted with the probability p_{add}. The inserted weight will be created when both units, the input and the output unit exists. Note that neglecting the selection there exists a equilibrium state of adding and deleting weights, i.e., the number of weights n in the network, that depends on the values of p_{add} and p_{del}: $n = \frac{p_{add}}{p_{add} * p_{del}}$.

Algorithm *simpleMut*:

for all (Weights of the reference net) **do**
  Search the appropriate weight in the offspring;
  if (the weight exists in the offspring) **then**
    if (RAND(0,1) < p_{del}) **then**
      delete the weight in the offspring;
  else if (the weight doesn't exist in the offspring) **then**
    if (both units, start and end-unit exist in the offspring) **then**
      if (RAND(0,1) < p_{add}) **then**
        insert weight in the offspring;

5.4.2 An other weight mutation

key word: **mutLinks**

`probadd` [p_{add}]
: This parameter x indicates the probability p_{add} for inserting a non existing weight.
Default: 0.0

`probdelStart` [$p_{delstart}$]
: This parameter x indicates the starting probability $p_{delstart}$ for deleting an existing weight. All probabilities between the $p_{delstart}$ (first generation) and p_{del} will be linear interpolated.
Default: 0.0

`probdel` [p_{del}]
: This parameter x indicates the height of the gaussian distribution for deleting an existing weight (See the algorithm *mutLinks*).
Default: 0.0

`sigmadel` [x]
: This parameter x indicates the width of the gaussian distribution for deleting an existing weight (See the algorithm *mutLinks*).
Default: 1.0

`probdelEndGen` [i]
: This parameter i indicates the generation in which the linear interpolation of the probability p_{del} for deleting an existing weight ends.
Default: 0

`initRange` [f]
: The parameter x determines the interval [-range, range] where inserted weights are randomly selected from.
Default: 0.5;

The module *mutLinks* executes a mixture of a simple weight mutation and pruning, called soft pruning. Each weight that exists in the reference net and not in the offspring net will be inserted with the probability p_{add}. The inserted weight will be created when both units, the input and the output unit exists. Each existing weight will be deleted by an gaussian distribution on the strength of the weight. This means that the probability to be deleted is for a small weight is greater than a bigger one.

Algorithm *mutLinks*:

for all Weights of the reference net **do**
  Search the appropriate weight in the offspring
  if (the weight exists in the offspring) **then**
    if (RAND(0,1) $< p_{del}^{t} e^{\frac{-weight^2}{sigmadel}}$) **then**
      delete the weight in the offspring
  else if (the weight doesn't exist in the offspring) **then**
    if (start and end-unit exist in the offspring) **then**
      if (RAND(0,1) $< p_{add}$) **then**
        insert weight in the offspring

5.4.3 Mutation of hidden neurons

key word: **mutUnits**

`probMutUnits` [x]
The parameter x indicates the probability p_{mut} a mutation takes place.
Default: 0.5

`probMutUnitsSplit` [x]
The parameter x describes the relationship between inserting and deleting hidden units.
Default: 0.5

`PWU` [x]
This switch activates the *Prefer Weak Units* strategy in the case of deleting hidden units.
Default: yes

`bypass` [x]
This switch turns on the *bypass* function while deleting a hidden unit.
Default: yes

`initRange` [x]
This parameter x describes the interval [-initRange,initRange] where inserted weights are randomly selected from. Default: 0.5

The module *mutUnits* executes a mutation of the hidden units. The maximum number of deleted or inserted hidden units is limited by one. All other mutation and optimization modules can only delete units, they can never insert weights to a deleted hidden unit. In this case of deleting an activated hidden unit, all weights will be deleted. In the case of inserting a hidden unit, all possible weights will be inserted. The number of weights and hidden units which can be inserted is limited by the topological structure of the reference net. The modules uses *bypass*-function and the *Prefer Weak Unit* strategy. The *bypass*-function is illustrated in figure 2. The idea is to linearly approximate a non-linear relationship, i.e., approximate the function computed by a hidden unit by a linear function. This is done by using shortcut (direct) connections. The *Prefer Weak Unit* strategy is to check the connectivity of a unit, i.e., count its incoming and outgoing weights, and prefer those for deleting which are weaker connected. This is called soft unit pruning.

Algorithm *mutUnits*:

if (RAND(0,1) > p_{mut}) **then**
    if (RAND(0,1) > p_{split}) **then**
        search a unit in the reference net which does not exist in offspring;
        insert this unit in the offspring;
        insert all possible weights of the unit;
    else
        if (PWU set) **then**
            select the *weakest* hidden unit of the offspring;
        else
            select accidental a hidden unit of the offspring;
        if (bypass activated) **then**
            delete the selected unit with the bypass function;
        else
            delete the unit and all its weights;

5.4.4 Mutation of the input units

key word: **mutInputs**

`probMutInputs` [x]
The parameter x indicates the probability p_{mut} that a mutation takes place.
Default: 0.5

`probMutInputsSplit` [x]
The parameter x indicates the relationship p_{split} between inserting and deleting of input units.
Default: 0.5

`initRange` [x]

The parameter x determines the interval [-range, range] where inserted weights are randomly selected from.
Default: 0.5

The module *mutInputs* executes a mutation only of the input units. The maximum number of deleted or inserted input units is limited by one. All other mutation and optimization modules can only delete units, they can never insert weights to a dead input unit.

The internal structure of SNNS does not allow to delete the input units like the hidden units, so the input units are just deactivated. In this case instead of deleting an activated input unit, all weights will be deleted. They are marked with the unit name **`xxx`**. If you analyze the network with the graphical user interface of SNNS, select in the display setup `show name` to easily identify removed input units. In the case of inserting a deactivated input unit, all possible weights will be inserted.

Algorithm *mutInputs*:

if (RAND(0,1) $> p_{mut}$) **then**
    if (RAND(0,1) $> p_{split}$) **then**
        insert all possible weights of the deactivated input unit
    else
        delete all weights of the activated input unit

5.5 Crossover

5.5.1 Crossover of the connections between input- and output layer

key word: **linCross**

`probCross` [x]

Probability of inserting a connection, that is contained in only one parent. Default: 0.5

This module does a linear crossover for all weights, which connect directly the input layer to the output layer, e.g., in nets without hidden layers or with shortcut connections. Only those connections are manipulated in the offspring net, no other units or weights are involved. If a network does not contain any of these connections, it remains unchanged.

Algorithm *linCross*:

delete all connections from all offsprings
forall possible connections **do**
  if (connection in both parents) **then**
    insert connection in offspring;
    set weight to the mean value of the parents' weights
  else if (connection only in one parent)**then**
    if (RAND(0,1) < p_{cross}) **then**
      insert connection in offspring;
      set weight to the value of the parent's weight;

5.5.2 Implant a feature from the fittest net in an offspring

key word: **implant**

`implantProb` [x]
  Probability of selecting a hidden unit of the first hidden layer from the best parent network and implanting it in one offspring network.
  Default: 0.2

This module selects a hidden unit from the first hidden layer in the fittest parent network and implants it in an offspring network. The hidden units are marked with their name to prevent implantation of a feature twice.

Algorithm *implant*:

repeat
  get hidden unit of the first layer in the fittest network;
  if (hidden unit does not exist in offspring net) **then**
    implant (add) unit to offspring net;
    forall (connections in parent network) **do**
      if (source unit does exist in offspring net) **then**
        insert connection with the same weight in offspring net;
until (a hidden unit was implanted or all hidden units failed)

5.6 Optimization

5.6.1 Learning stopped by periods or learning error

key word: **learnSNNS**

`learnfct` [x]
: This parameter x contains the name of the SNNS-learning function.
Default: Rprop

`learnparam` [x]
: This array of parameters indicates the parameter for the SNNS learning function. The meaning of the parameters can differ from learning function to learning function. For details please see the SNNS manual.
Default: 0.0 0.0 0.0 5.0 0.0

`maxepochs` [x]
: This parameter x contains the maximum number of periods for the learning algorithm. After this maximum the module *learnSNNS* will automatically stop the learning function.
Default: 50

`maxtss` [x]
: This parameter indicates the maximal tolerable learning error of the network. The error is normalized by the number of learning patterns and the number of output units. The module *learnSNNS* will terminate the learning function if the learning error is less then this threshold.
Default: 0.5

`shuffle` [x]
: This switch indicates whether the sequence of the learning patterns is changed after each learning period or not. If the switch is turned on, then the module *learnSNNS* will use the SNNS-function *shuffle*
Default: yes

The module *learnSNNS* is the standard-learn module during the optimization. The module stops the learning of an offspring network if the learning error is below an upper bound or it reaches the maximum number of learning periods.

Algorithm *learnSNNS*:

Set the SNNS learn function *learnFct*;
for all (networks of the offspring population) **do**
  while (Epochs $<$ *maxEpochs*) **and** (tss $>$ *maxTss*) **do**
    Train one period with *learnParam*;

5.6.2 Learning stopped by periods or cross validation error

key word: **learnCV**

`learnfct` [x]
: This parameter x contains the name of the SNNS-learning function.
Default: Rprop

`learnparam` [x]
: This array of parameters indicates the parameter for the SNNS learning function. The meaning of the parameters can differ from learning function to learning function. For details please see the SNNS manual.
Default: 0.0 0.0 0.0 5.0 0.0

`maxepochs` [x]
: This parameter x contains the maximum number of periods for the learning algorithm. After this maximum the module *leranSNNS* will automatically stop the learning function.
Default: 50

`CVepochs` [x]
: This parameter x sets after how often the error on the cross validation set for cross validation is computed.
Default: 2

`shuffle` [x]
: This switch indicates whether the sequence of the learning patterns is changed after each learning period or not. If the switch is turned on, then the module *learnCV* will use the SNNS-function *shuffle*
Default: yes

The module *learnCV* uses the error an a cross validation pattern set to stop learning. If it increases again learning is stopped. The module stops the learning of an offspring network if the error on a cross vailidation set starts increasing again or it reaches the maximum number of learning periods. This is done by computing the error each *CVepochs*. If the current error is larger than the average of the last four values, learning is stopped.

Algorithm *learnCV*:

Set the SNNS learn function *learnFct*;
for all (networks of the offspring population) **do**
  while (Epochs $<$ *maxEpochs*) **and** ($tss(t) < \sum_{k=t-4}^{k=t-1} tss(k)$) **do**
    Train one period with *learnParam*;

5.6.3 Delete all weights below a threshold

key word: **prune**

`threshold` [t]

The parameter t indicates the threshold, underneath all weights of the net will be deleted.
Default: 0.0

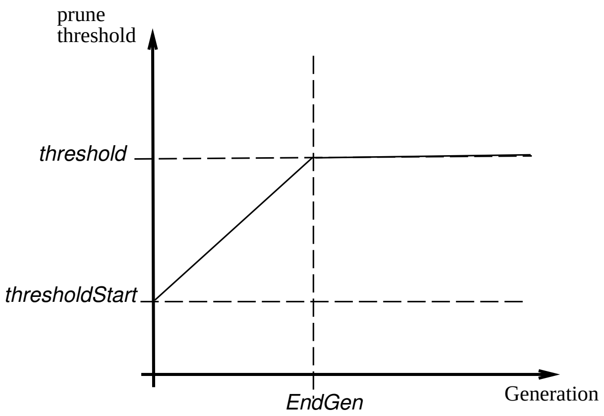
`thresholdStart` [f]

The parameter x indicates the start-threshold for the pruning module.
Default: 0.0

`pruneEndGen` [i]

The parameter i indicates th number of the generation until the pruning module will take the origin *threshold* .
Default: 0

All weights of the net with an abolute strength under the threshold will be deleted.



5.6.4 Adaptive pruning

key word: **adapPrune**

`threshold` [x]

This value gives the threshold used to initialize the parents.
Default: 0.0

`deltaThreshold` [x]

This value gives the maximal factor the threshold of the offsprings is allowed to differ from the parents.
Default: 0.2

`aveThreshold` [x]

For every network the mean value of the absolute value of the weights is computed and the threshold is individually set to this value times *aveThreshold*.
Default: 0.0

The module *adapPrune* is an adaptive variation of standard pruning. Weights which are smaller than the threshold are deleted. The threshold is set individually for every network, depending on the mean value of its weights. If the factor *aveThreshold* is not set the value *threshold* is used. The distribution of possible changes is realized with a Gaussian distribution $g(x)$ and a maximal factor:

$$(\Delta_{threshold} = g(x) * \ deltaThreshold).$$

5.6.5 Relearning

key word: **relearn**

`relearnfactor` [*f*]
: The parameter x indicates the factor with which all weights and biases will be multiplied with.
Default: 1.0 (no change of the weights and biases takes place.)

In the mind of *lamarckism* not only the topology of the parents will be transmitted to the offspring, but also the strength of connections inside the structure. In the case of neural networks this will lead to a local minimum in the learning function. To escape from this local minimum it is necessary to change the weights a little bit. The module *relearn* multiplies all weights and biases with a given factor.

5.6.6 Adding random distributed values

key word: **jogWeights**

`jogLimit` [*f*]
: The range of values added is given by $[-Paramf, Paramf]$. Default: 0.01

In the mind of *lamarckism* not only the topology of the parents will be transmitted to the offspring, but also the strength of connections inside the structure. To get out of the centre of a local minima the module *jogWeights* adds random uniform distributed values to the connection weights. This is an alternative to the *relearn*-module, that multiplies all weights by a constant factor.

5.6.7 Cleanup the structure of a net

key word: **cleanup**

`no parameters`

The module *cleanup* deletes all units and weights which have no direct or indirect connection to the input or output layer of the net.

5.6.8 Delete offsprings without links

key word: **nullWeg**

`no parameter`

The module *nullWeg* deletes all offsprings which contain no connections. This could otherwise eventually lead to misbehavior in other modules.

5.7 Evaluation

5.7.1 Evaluation of the topology

key word: **topologyRating**

`weightRating` $[f]$
: The number of connections in the network is multiplied by this value and divided through the maximal number of connections (in the reference network). The result is added to the fitness term.
Default: 0.0

`unitRating` $[u]$
: The number of hidden units in the network is multiplied by this value and divided through the maximal number of hidden units (in the reference network). The result is added to the fitness term.
Default: 0.0

`inputRating` $[k]$
: The number of input units in the network is multiplied by this value and divided through the maximal number of input units (in the reference network). The result is added to the fitness term.
Default: 0.0

The module *topologyRating* evaluates the topology of all offspring networks. Criteria are the number of connections, the number of hidden units and the number of input units. These numbers are multiplied by a scaling factor and divided through the maximal values of the reference network. The result contributes to the fitness.

5.7.2 Evaluation of the learning process

key word: **learnRating**

`noLearnRating` $[f]$
: This value is added to the fitness term if the network couldn't learn the patterns properly, e.g. its mean error was higher than specified by *maxtss*
Default: 200.0

`epochRating` [*f*]
The number of learning epochs needed is multiplied by this value and added to the fitness term.
Default: 0.0

`tssRating` [*f*]
The mean error is multiplied by this value and added to the fitness term.
Default: 0.0

`maxtss` [*f*]
If the mean error is lower than this value, learning is stopped. It is necessary to decide if learning was successful or not.
Default: 0.5

The module *learnRating* evaluates the learning properties of all offspring networks. It is possible to evaluate the mean error, the number of training epochs until the mean error is lower than a given threshold and additionally punish networks which couldn't learn the patterns with a specified precision.

5.7.3 Evaluation using the 40 - 20 - 40 method

key word: **classes**

`crossPattern` [*name*]
Name of the file which contains the set of cross validation patterns.
Default: -

`hitRating` [*f*]
Value that is added to the fitness term, in case the network classifies a pattern correctly.
Default: 0.0

`missRating` [*f*]
Value that is added to the fitness term, in case the network classifies a pattern wrong.
Default: 10.0

`noneRating` [*f*]
Value that is added to the fitness term, in case the network doesn't classifies a pattern.
Default: 10.0

`highDesc` [*f*]
Maximal output value of the output neuron.
Default: 1.0

`lowDesc` [*f*]
Minimal output value of the output neuron.
Default: -1.0

`decisionThreshold` [*f*]
Distance between the output of *min-activation* and the output of *max-activation*.
Default: 0.2

The module is useful for two state classification networks with one output neuron, i.e., one value reflecting a positive classification (usually 1), and the other reflecting the negative classification (usually 0). The values are set by *highDesc* and *lowDesc*. The distance *decisionThreshold* is taken around the average. If the output is within the *decisionTreshold* area it is taken as not classified.

5.7.4 Evaluation using the highest output

key word: **bestGuessHigh**

`crossPattern` [*name*]
Name of the file which contains the set of cross validation patterns.
Default: -

`hitRating` [*f*]
Value that is added to the fitness term, in case the network classifies a pattern correctly.
Default: 0.0

`missRating` [*f*]
Value that is added to the fitness term, in case the network classifies a pattern wrong.
Default: 10.0

`noneRating` [*f*]
Value that is added to the fitness term, in case the network doesn't classify a pattern.
Default: 10.0

`hitThreshold` [*f*]
The value gives the threshold that needs to be reached, before classifying takes place
Default: 0.3

`hitDistance` [*f*]
The value gives the distance between the output of two neurons, before a classification counts as valid. Default: 0.2

The module *bestGuessHigh* tests the generalization performance of a network and increases the fitness. It is useful for a Winner-takes-all output (1 out of n) properties. The neuron with the highest activity is selected as winner. If its activity distance to the next highest activated neuron is smaller than *hitDistance* the pattern is treated as not classified.

5.7.5 Evaluation using the lowest output

key word: **bestGuessLow**

`crossPattern` [*name*]
Name of the file which contains the set of cross validation patterns.
Default: -

`hitRating` [*f*]
: Value that is added to the fitness term, in case the network classifies a pattern correctly.
Default: 0.0

`missRating` [*f*]
: Value that is added to the fitness term, in case the network classifies a pattern wrong.
Default: 10.0

`noneRating` [*f*]
: Value that is added to the fitness term, in case the network doesn't classifies a pattern.
Default: 10.0

`hitThreshold` [*f*]
: The Value gives the threshold that needs to be reached, before classifying takes place
Default: 0.3

`hitDistance` [*f*]
: The Value gives the distance between the output of two neurons, before a classification counts as valid.
Default: 0.2

The module *bestGuessLow* tests the generalization performance of a network and increases the fitness. The neuron is selected which has the lowest activity, e.g., a Looser-takes-all selection. If its activity distance to the next lowest activated neuron is smaller than *hitDistance* the pattern is treated as not classified.

5.7.6 Evaluation through a cross validation set

key word: **tssEval**

`crossPattern` [*name*]
: Name of the file which contains the set of cross validation patterns.
Default: -

`crossTssRating` [x]
: factor for multiplying the mean error per pattern on the cross validation set. Default: 0.0

`crossHamRating` [x]
: Value to add for each wrong classified pattern. Default: 0.0

`crossHamThresh` [x]
: possible distance, that is allowed for the output from the target. A pattern is wrong classified, if at least the output of one output neuron differs from its target by more than [x]. Default: 0.0

This module computes a simple fitness term on a cross validation set. The mean error per pattern as well as the classification performance can be evaluated.

5.8 History

5.8.1 A simple version of saving all important informations

key word: **histSimple**

`historyFile` *filename*
This string contains the prefix of the filename where all informations are stored. The extension of the filename is *.simple* .

A simple record that writes down all needed informations about the network, e.g. learning, topology and fitness values. Each row contains the informations about one network.

5.8.2 Saving all informations about fitness of the networks

key word: **histFitness**

`historyFile` *filename*
This string contains the prefix of the filename where all informations are stored. The extension of the filename is *.fit* for the net informations and *.popfit* for the population informations.

The modul *histFitness* writes all fitness informations of a net in a special file (extension *.fit*). Each line of the file accords to one net.

The file with the *.popfit* extension contains informations about the fitness of whole population. Each line describes the several fitness values (best fitness of all members, worst fitness of all members and the average fitness) of the population at each generation. The population file is ready for gnuplot and other programs.

The fitness values are not computed in this modul, they are computed during the *evalution* functions and stored in special slots of the data structure.

5.8.3 Saving all topology informations

key word: **histWeights**

`historyFile` *filename*
This string contains the prefix of the filename where all informations are stored. The extension of the filename is *.weight*.

A simple record that writes down all needed informations about the topology of the network, number of weights, number of hidden units and number of active input units. Each row contains the informations about one network.

5.8.4 Saving all informations about the networks on the cross validation patterns

key word: **histCross**

`historyFile` *filename*
: This string contains the prefix of the filename where all informations are stored. The extension of the filename is *.cross* for the network informations and *.popcross* for the population informations.

The modul *histCross* writes all informations about the network in a special file (extension *.cross*) Each line of the file accords to one network. It contains information about the number of hits (the network classified the pattern correct), miss (the network cassified the pattern wrong) and nones (the network did not classify the pattern in an unique way). It also contains the quotient hits and misses.

The file with the *.popcross* extension contains informations about the whole population. Each line describes the several test values (best quotient of all members, worst quotient of all members and the average quotient) of the population at each generation. The population file is ready for gnuplot and other programs.

The values are not computed in this modul, they are computed in the evalution functions and stored in special slots of the data structure.

This module is only useful in conjunction with evaluation modules using classification performance, e.g. *bestGuessHigh* or *classes*.

5.8.5 Family tree

key word: **ancestry**

`historyFile` [*histfile*]
: Prefix for the output file
Default: enzo.hst

`ancestryPS` [x]
: If the flag is set a postscript figure of the family tree is generated.
Default: NO

This module writes the `histIDs` of each generation. It is possible to see when each network is born and how long it survives in the population. Offsprings which never enter the population are not shown. It's possible to generate a postscrip figure from the family tree by setting the flag to `YES`.

5.8.6 Simple X-Window history

key word: **Xhist**

`Xgeometry` [x] [y] [*width*] [*height*]
Specifies the size and position of the window on the terminal. The top left corner is given by [*(x,y)*]. the width and height by [*width*] and [*height*].
Default: 10 10 600 300

`Xcoord` [*xll*] [*yll*] [*xur*] [*yur*]
Defines the coordinate system for the graphical representation of the fitness values. The left lower corner is specified by [*(xll, yll)*] and the right upper corner by optParam(xur, yur).
Default: 0.0 0.0 30 1000.0

A X-Window with a coordinate system is shown. The best, average and worst fitness values are plotted vs. generation number. The size of the window and the coordinates are adjustable by the user. This module uses the `xgraf` program, located in the directory `ENZO/src/history/Xgraf`. To compile xgraf type `make xgraf` in the `ENZO` directory.

5.8.7 Used input units

key word: **histInputs**

`historyFile` [x]
This string contains the prefix of the filename of the output file.
Default: `enzo`

The modul *histInputs* writes for each input unit of a net whether it is active or not. The file for the output is named by the prefix *historyFile* and the extension *.inputs* .

The information for each network consist of one row, where all the informations are written down. A D means that this input unit is absent in the net, a X means that the input unit is active.

5.9 Survival

5.9.1 Survival of the fittest

key word: **fittestSurvive**

`NoOfOffsprings` k
This parameter gives the number of offsprings which are to be inserted in the population. offsprings k.

This module inserts better offsprings into the population and removes worse parents. The individuals are sorted by their fitness, the lower the better.

5.10 Post-evolution

5.10.1 Storing networks after evolution

key word: **saveAll**

`netDestName` *filename*
Sets the prefix of the filename. Networks are written to files with the prefix followed by a number for each network and the suffix .net, e.g. *filename*`_1.net` .

`saveNetsCnt` [*cnt*]
Number of networks, which are to be stored.
Default: 99

Saves the given number of networks at the end of the evolution. Usually this are parent networks and the reference net. The networks are numbered with increasing fitness, e.g. the best network is *filename*`_0.net`

5.11 Sample module

5.11.1 My_module title

key word: **mymodule**

`initialize` [x]
Description of this parameter.
Default: -

`exit` [x]
Description of this parameter.
Default: -

`myParam` [x]
Description of this parameters.
Default: -

Here should follow a description of the functionality of the whole module: What is it for, when to use an when not to use it, etc.

6 Adjusting parameters

You should not be worried about the amount of adjustable parameters. Most of same are easy to handle, in a way that they have sensible default values and modifications have little influence on the result. Still for some problems it might be useful to have the opportunity to tailor the algorithm in a certain way.

Some parameters need to be set in an intelligent way, i.e., you should take time and use your knowledge about the problem to adjust them.

Firstly, the parameters of the local optimization depend heavily on the problem. That is the mean error *maxTss* and the number of epochs *maxEpochs* should be set to values that provide a good solution. Note that since (in case of our mutation operators) offsprings have some knowledge of their parents, they need to be trained significantly less[1]. If weight decay is used, it should be adjusted ina a way that no overfitting occurs.

Secondly, the design of the fitness function is important, because that's our optimization criteria. You should compute all fitness terms for your reference net (the maximal topology) and give those higher weights, that you care about. Be aware that some constraints are maintained, e.g., if a network can't learn the training patterns, its fitness should reflect this clearly.

The size of the population, the number of offsprings and the number of generations should be in a sensible range. The more generations the better the result (with respect to your fitness function !). The bigger the population and the higher the number of offsprings created in each generation, the wider the exploration. For a given amount of time, you need always to decide the relation of exploration to exploitation, e.g. creating many offspring each generation vs. creating fewer offsprings but use more generations.

Sensible values are, if possible, at least 30 generations, 30 networks in the parent population and creating 10 offsprings each generation.

The probabilities of mutation should be set in a way that at the most 1% to 10% of the links resp. units are mutated. Otherwise the offsprings will loose most of the knowledge of the parents.

Acknowledgements

Several students made valuable contributions to the development of ENZO by studiing the evolution of neural networks in their master thesis (Weisbrod, 1992, Zagorski, 1994, Schäfer, 1994, Schubert, 1995). This implementation of ENZO goes back to the work of (Schäfer, 1994, Schubert, 1995).

[1] Since the networks usually are smaller and due to the knowledge transformation they possible speedup is in the range from 10 to 50.

Part II

ENZO Implementation Guide

7 Design

The kernel of ENZO has a simple structure:

```
init-modules

read-control-file

{pre-evolution}

{evaluation}
{history}
{survival-of-the-fittest}

while not {stop-evolution} do
begin
    {selection}
    {crossover}
    {mutation}
    {optimization}
    {evaluation}
    {history}
    {survival-of-the-fittest}
end
{post-evolution}

exit-modules
```

The names in brackets correspond to the evolutionary operators, which consist of several modules. All modules of one operator type are linked to a library, e.g., each operator has a corresponding library. This allows easy extension, and since all functionality is implemented in the libraries, one can use the skeleton implementation of the algorithm to optimize other things than neural networks by exchanging the libraries. The flexible design also allows for a good parallelization, if necessary.

ENZO was implemented in ANSI-C to allow portability to other systems. On the other hand it would be very appealing to implement a genetic algorithm (as well as neural network simulators) in a object oriented language like C++. To achieve some of the flexibility of an object-oriented design, all modules have a standardized interface, and their functions are referred through function pointer, stored in a module table.

7.1 Extension of Enzo by own modules

For extension exists a sample module, which should allow you to integrate an own module specialized for your purposes. Just try it ...

To add an own module, the following things have to be done:

1. Design of the interface functions.
2. Implementation of the module.
3. Test the module extensively!
4. Add the new module in the global module table, and change the makefile such that your code is compiled and linked to the program.
5. Don't forget the documentation of key words and functionality of the module.

7.2 Dependencies between modules ?

Ideally, there should be no dependencies !

Some kind of dependencies between certain operators seem still suggestive, e.g. during pre-evolution one might also want to perform a local optimization. The corresponding module is part of the optimization library. By sharing the keywords[2] the pre-evolution module can find the optimization function through the module table and call it via the module interface. To provide this functionality a module should indicate if it wants to use a command line exclusively, i.e., it should be removed and not further dispatched.

7.3 Sharing data

The nepomuk library provides a user data field additionally to the network information, e.g., the structure `networkData` is used to store the data of the evolution process with the individuals. This is necessary to append the fitness to the network, but it could also be used to optimize the parameters of the evolution ! One should be careful with this global-like data. Before adding a new field to this structure, alternatives that do not need global data exchange should be evaluated.

8 Interfaces

There are three levels of interfaces to distinguish: the standardized module interfaces, the nepomuk interface and the interface to the neural network simulator. They are discussed in the following sections.

[2] This is similar to a message dispatcher in a window-based environment

8.1 Module interfaces

Every module has to implement the following functions,which are made public through the module table. Also has each module to define a keyword, which activates the module during initialization. The syntax is simply: `keyword` .

`int module_init( ModuleTableEntry *self, int msgc, char *msgv[] )`

This function handles the initialization of the module.
`message = "initialize"` : for general initialization;
`message = "exit"` : for general exit;

`other keywords` : specific initialization.
`self` the entry in the global module table.

`return value:`
0 : The message was not used.
1 : The message was used.
< 0 : A warning should be generated.
> 1 : An error message should be generated.

`int module_work( PopID *parents, PopID *offsprings, PopID *ref )`

This is the working horse of every module.
These functions should not manipulate neither the parents nor the reference networks, but only the offsprings. During selection, the concerning networks are copied to an offspring population, which can be manipulated.

`char *module_errMsg( int err_code )`

Returns a error message depending on `err_code` .

The activation of modules is done with the function `enzo_actModule()` . Furthermore the function `enzo_logprint( char *fmt, ... )` allows for output in the log file (as `printf()`).

8.2 The Nepomuk library

All Nepomuk interface functions have the prefix `kpm_`, shortcut for **K**arlsruher **P**opulations-**M**anager;

Initialization and cleanup of Nepomuk :

`kpm_err kpm_initialize( int max_nets, int max_pats )`

Initializes the internal data structures and fields of Nepomuk . The number of networks to manage is given by `max_nets` and the number of pattern sets by `max_pats`.

`kpm_err kpm_exit( void )`
For a clean exit this function needs to be called. It frees allocated memory and resets NEPOMUK .

Network management:

`kpm_err kpm_setCurrentNet( NetID id )`
Selects the network with the given `id` for further manipulation.

`NetID kpm_getCurrentNet( void )`
Returns the `NetID` of the currently activated network.

`NetID kpm_loadNet( char *filename, void *usr_data )`
Loads a new network from the file `filename` , makes it the active network and returns its `NetID` . The given `usr_data` is appended to the structure.

`kpm_err kpm_saveNet( NetID id, char *filename, char *netname )`
Saves the network referred by `id` in the file `filename`.

`kpm_err kpm_deleteNet( NetID id )`
Deletes the complete data structures of the network referred by `id`, i.e., the internal representation as well as the `userData`.

`kpm_sortNets( CmpFct netcmp )`
Sort the networks in NEPOMUK using the function `netcmp`. The function works as `strcmp` and keeps the networks sorted. Is `netcmp == NULL` no sorting is done.

`NetID kpm_copyNet( NetID id, void *usr_data )`
Creates a copy of the network, appends the given `usr_data` and returns the `NetID`.

`NetID kpm_newNet ( void *usr_data )`
Creates a new (empty) network, appends the given `usr_data` and returns the `NetID`.

`void *kpm_getData( NetID id )`
Returns the `usr_data` of the active network.

`kpm_err kpm_getNetDescr( NetID id, NetDescr *n )`
Returns a description of the active network, e.g. number of units, weights, etc.

Management of subpopulations:

`PopID kpm_newPopID( void )`
Returns a new, unused population identification.

`kpm_err kpm_validPopId( PopID id )`
Checks, if the identification is valid. Returns `KPM_NO_ERROR` if valid and `KPM_INVALID_POPID` otherwise.

`kpm_err kpm_setPopMember( NetID n_id, PopID p_id )`
Makes the network referred by `n_id` a member of the population referred by `p_id` .

`PopID kpm_getPopID( NetID id )`
: Returns the identification of the population the network referred by `n_id` belongs to.

`NetID kpm_popFirstMember( PopID p_id )`
: Returns the network identification of the first network in the population referred by `p_id`. If no network exists, `NULL` is returned, otherwise the network is made the active network. The first network is the smallest network with respect to function passed to `kpm_init` .

`NetID kpm_popNextMember( PopID p_id, NetID n_id )`
: Returns the network identification of the network direct after `n_id` in the population `p_id`. If none exists, `NULL`, else the network is mad the active network.

Pattern management:

`PatID kpm_loadPat( char *filename, void *usr_data )`
: Loads a new pattern set from the file `filename`, appends the `usr_data` and returns its `PatID` .

`kpm_err kpm_setCurrentPattern( PatID id )`
: Sets the pattern set referred by `id`.

`PatID kpm_getCurrentPattern( void )`
: Returns the `PatID` of the current pattern set.

`void *kpm_getPatData( PatID id )`
: Returns the `usr_data` of the pattern set referred by `id`.

`PatID kpm_getFirstPat( void )`
: Returns the `PatID` of the first pattern set, or `NULL` if none exists.

`PatID kpm_getNextPat( PatID pat )`
: Returns the `PatID` of the pattern set following the set referred by `pat` , or `NULL` if none exists.

`kpm_err kpm_setPatName( PatID id, char *name )`
: A name can be assigned to pattern set for identification, e.g., "'learn"', "'test"', "'cross validation"', ... This functions copies the given string to store it with the pattern set.

`char *kpm_getPatName( PatID id )`
: Returns the name of the pattern set referred by `id` or `NULL` if none exists.

8.3 Interface to a neural network simulator

All interface functions have the prefix `ksh_`. They are located in the file `kr_shell.c` which is linked to the nepomuk library. There are about 100 interface functions for network manipulation request, e.g., to delete units, to add units, to perform learning, etc...

Every simulator that offers a large set of manipulation functions, as the SNNS simulator does, could be used with Enzo . The functions of the interface do almost nothing except calling the appropriate functions of the neural network simulator.

9 Implementation internals

9.1 Enzo

Enzo is no stand alone program, but uses a neural network simulator for network manipulation. It provides an interface to the simulator that should allow for easy communication with most simulators, that provide an interface for network manipulation. Figure 4 shows the design and the interfaces of Enzo in connection with the SNNS simulator, figure 7 shows which libraries and objects are linked together to the executable `enzo`.

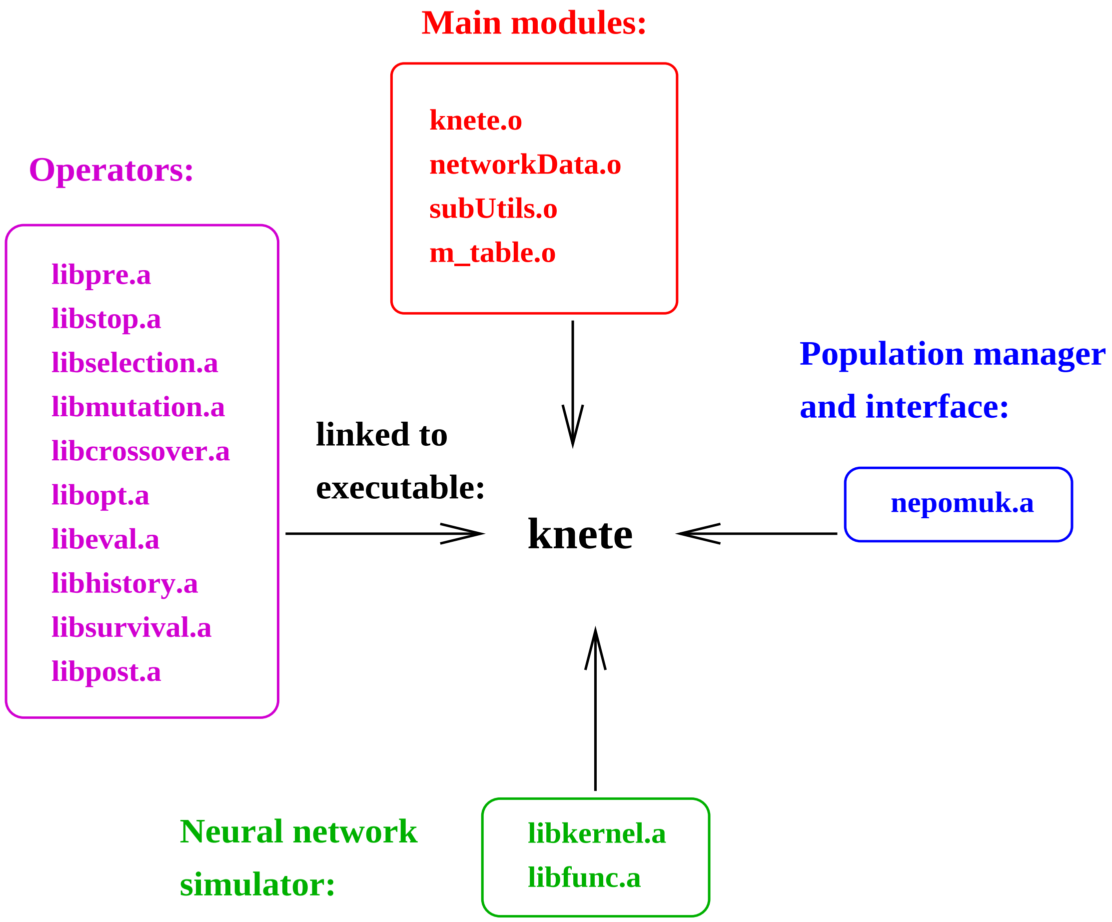


Figure 7: The figure shows all libraries and objects which are linked together to form the executable `enzo`.

The interface functions of the simulator are called solely through Enzo 's interface module (kr_shell.c). These interface functions are, in turn, called from the modules forming the evolutionary operators or from the population manager Nepomuk . The handling of networks and patterns is done exclusively via the Nepomuk interface. The main loop is linked with the other parts through function pointers. It uses a module table to store all functions, and activates dependent on the command file the specialized modules. The function pointers forming an operator are stored in a corresponding table. Note that a pointer could appear here several times.

Summarizing, ENZO consists of three main parts: the main loop, the population manager library NEPOMUK and the specialized modules forming the evolutionary operators libraries.

9.2 NEPOMUK

The population is managed with an array of constant size; the number of population members is fixed at initialization. Note that the functions for subpopulation management are designed for rather small population (linear search), i.e., several hundred members would be impossible. This would also lead to a memory problem.

For an efficient population management, free and used array elements are stored in lists, i.e., requirements can be satisfied in O(1). See also figure 8.

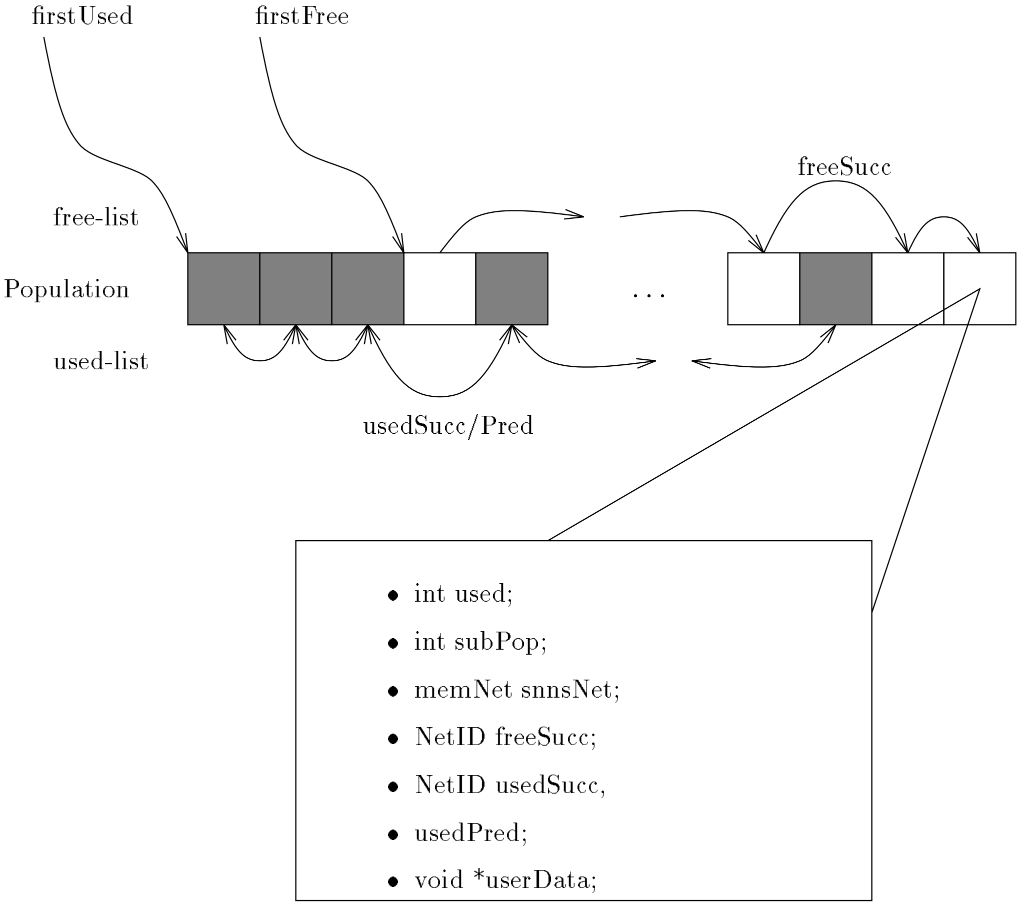


Figure 8: *Data structure used in* NEPOMUK .

9.3 Necessary changes in SNNS modules

Two little changes had to be made to the original SNNS software. Since we are working with a population of networks, the allocated memory for units, etc... should be in a descent range. For that, the default block size for memory allocation in the file **kr_def.h** were decreased.

Furthermore, the handling of several networks needs additional interface functions to copy network information into the data structures of the SNNS kernel. The functions located in the file `enzo_kr_mem.c` provide this functionality. It's included in the SNNS file `kr_mem.c`. To activate this extensions you have to compile the SNNS kernel library with the flag `-D__ENZO__`.

A An example command file

```
#
# sample command-file for tc-problem                       --- TR, 22.08.95 ---
#



#########################
#       files           #
#########################


network      tc.net        # name of network-file
learnpattern tc.pat        # name of learnpattern-file
testpattern  tc.pat        # name of testpattern-file

historyFile  hst           # append this suffix to history files
netDestName  tc_erg        # save nets with this prefix


#########################
#       modules         #
#########################


# ----- initialize and pre-evolution -----------------------------------------

genpopNepo     # initialize nepomuk
loadSNNSPat    # load SNNS-Pattern
initPop        # create and initialize Population
optInitPop     # optimize initial population

# ----- stop-evolution -------------------------------------------------------

stopErr        # stop if something's wrong with parents/offsprings
stopIt         # stop after maxGenerations

# ----- selection -------------------------------------------------------------

preferSel      # use random prefer selection

# ----- mutation --------------------------------------------------------------

simpleMut      # do simple mutation
mutUnits       # delete or add hidden units
mutLinks       # delete or add links
mutInputs      # cut off input units

# ----- optimize -------------------------------------------------------------

nullWeg        # delete useless nets
```

```
relearn        # scale weights before relearning
learnSNNS      # do learning via SNNS-Function
prune          # use pruning
cleanup        # remove dead units
nullWeg        # delete useless nets
learnSNNS      # do re-learning via SNNS-Function

# ----- evaluate -------------------------------------------------------------

learnRating    # increase fitness if training patterns were not learned
topologyRating # increase fitness dependent on topology

# ----- history ---------------------------------------------------------------

histSimple    # write standard information
histFitness   # write fitness information
histWeights   # write weight and unit information
Xhist         # show fitness in a X-Window

# ----- survival --------------------------------------------------------------

fittestSurvive  # sort nets by fitness (the lower the better)

# ----- post-evolution --------------------------------------------------------

saveAll         # save networks


#########################
#     parameters        #
#########################



maxGenerations 20   # stop after  x  generations

popsize   30        # population size
gensize   10        # new members each generation

preferfactor 3.0    # preferfactor for better nets


# ----- optimization --------------------------------------------------------

learnModul learnSNNS    # use standard SNNS-learning


learnfct   Rprop                  # learning function to be set
learnparam 0.00 20.99 0.0 5.0 0.0 # and parameters
maxtss     0.01                   # stop learning if error is smaller
maxepochs  100                    # stop after x epochs
shuffle 1                         # shuffle patterns
relearnfactor 0.5                 # scale weights before relearning by x
```

```
threshold 0.2                     # minimal treshhold for pruning
thresholdStart 0.5                # Start with this treshold
pruneEndGen 30                    # use minimal treshold from this generation



# ----- mutation -------------------------------------------------------------

probadd 0.2              # probability of adding a link
probdel 0.1              # probability of removing a link

probMutUnits 0.1         # probability of unit mutation
probMutUnitsSplit 0.9    # probability relation between adding and deleting

probMutInputs      0.6   # probability of input unit mutation
probMutInputsSplit 0.6   # probability relation between adding and deleting



# ----- evaluation -----------------------------------------------------------

                         # linear sum of:
weightRating   20.0      # number of weights times x
unitRating     20.0      # number of units times x
inputRating   200.0      # number of inputs units times x
tssRating      30.0      # mean error  times x
noLearnRating 200.0      # add x if training patterns were not learned


# ----- history --------------------------------------------------------------

Xgeometry 650 10 600 330   # size and position of X-Window
Xcoord 0.0 0.0 20.0 600.0  # size of the coordinate system


# ----- post-evolution -------------------------------------------------------

saveNetsCnt 5            # save x nets

# eof
```

Index

References

Braun, Heinrich, Johannes Feulner, and Volker Ullrich. In *Neuro Nimes 91*.

Goldberg, David (1989) *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.

McDonell, John and Don Waagen (1993). Neural structure design by evolutionary programming. NCCOSC, RDT&E Division, San Diego, CA 92152.

Reeves, Colin (1993) *Modern Heuristic Techniques for Combinatorial Problems*, volume 1. Orient Longman.

Riedmiller, Martin and Heinrich Braun (1993) A direct adaptive method for faster backpropagation learning: The RProp algorithm. In *Proceedings of the ICNN 93*, San Francisco.

Schäfer, Johannes (1994) Evolution Neuronaler Netze zur Erkennung von handgeschriebenen Ziffern. Diplomarbeit, Univerität Karlsruhe, Institut für Logik Komplexität und Deduktionssysteme.

Schubert, Matthias (1995) Evolutionäre Optimierung Neuronaler Netze zur Zeitreihenvorhersage. Diplomarbeit, Univerität Karlsruhe, Institut für Logik Komplexität und Deduktionssysteme.

Schwefel, Hans-Paul (1995) *Introduction to the theory of neural computation.* Santa Fe Institute, Studies in the sciences of complexity, lecture notes. Addison-Wesley.

Weisbrod, Joachim (1992) Einsatz Genetischer Algorithmen zur Optimierung der Topologie mehrschichtiger Feedforward-Netzwerke. Diplomarbeit, Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme.

Zagorski, Peter (1994) Entwicklung Evolutionärer Algorithmen zur Optimierung der Topologie und des Generalisierungsverhaltens von Multilayer Perceptrons. Diplomarbeit, Univerität Karlsruhe, Institut für Logik Komplexität und Deduktionssysteme.