BAKKALAUREATSARBEIT

Design and Implementation of a Generic Framework for Genetic Optimization of Neural Networks

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Bakkalaureus der Medizinischen Informatik

unter der Leitung von

Univ.Ass. Dipl.-Ing. Dr.techn. Wilfried Elmenreich
Institut für Technische Informatik 182

durchgeführt von

Andreas T. Pfandler
Matr.-Nr. 0325859
Favoritenstraße 139/7, A-1100 Wien

Wien, im September 2006 .....

Design and Implementation of a Generic Framework for Genetic Optimization of Neural Networks

Neuronal networks in combination with genetic algorithms provide a flexible method for solving various problems (Especially when we only can define a fitness function for a solution, rather than giving a feasible, exact solution). To be able to work with this concepts in an efficient way an extensible, highly customizable framework is proposed.

This work starts with a short historical overview and then presents the basic concepts of neuronal networks and genetic algorithms. Some more advanced concepts like meta-evolution, genetic variance and parallel processing of fitness evaluation will discussed. Those concepts may help to improve the performance of the methods used in the framework. Some important features of the Neuronal Network and Genetic Algorithms (NNGA) framework will be dealt with. The work will deal examine the performance of the developed framework and propose some improvements. In the appendix the interested reader will find a short setup / user guide for NNGA.

Contents

1. Introduction

This work will describe the concepts and the background which were relevant for developing the Neuronal Network and Genetic Algorithms (NNGA) framework software. This framework will be licensed under the Gnu Public License (GPL) so that it's easier for other people to do research in this field or to improve the framework.

This framework tries to take advantage of two concepts:

- neuronal networks
- genetic algorithms

Although they seem to be completely different they have their origin in common: Both concepts were inspired by nature. Together those two concepts can be used to solve various problems.

Neuronal networks are a quite old concept. Their concept was introduced by McCulloch and Pitts in 1943. The article can be found in [MP43]. Donald Hebb published his idea of learning neurons in 1949 [Heb49]. In 1958 Rosenblatt proposed the model of the perceptron. Many other researchers based their work on those articles.

Genetic Algorithms were inspired by evolution. For historic information on genetic algorithms this work refers to [Fog00] (chapters 3.5 and 3.6).

So we can see that neuronal networks and genetic algorithms are a powerful combination for optimizing and evolving neuronal networks.

The aim of the developed software is to provide a possibility to work with those two concepts in a easy and flexible way. Thereby other researchers can easily test their modifications and check if they bring an advantage. Some ideas which can be integrated into the framework will be discussed later. The NNGA framework can be used to develop strategies or control software for problems which are difficult to calculate. Even in the research field of real time systems neuronal networks can help solving complex problems.

1.1. Motivation and Objectives

The goal of the Project is to provide a framework which can be used for research. It will be easy to integrate new features or modify existing ones. Features that

have not been implemented yet will be easy to implement later. This work will also describe the concepts used in NNGA framework so that the reader is able to use it afterwards. The framework can be used to develop game strategies for many different game types. Results gained from work with the framework can be integrated in robot control software. These are only two of countless other examples which show that the framework can be used for different types of research.

1.2. Structure of the Thesis

The thesis is structured as follows: Chapter 2 gives an introduction into neural networks and genetic algorithms which are implemented in the NNGA framework.

Chapter 3 will discuss the used libraries, the design and the ideas behind the program.

Based on this design Chapter 4 will describe the implementation, the implementation problems and how these problems were solved.

In Chapter 5 the results of some performance tests will be shown and discussed. Furthermore some possibilities to increase performance will be talked about.

Finally, the thesis ends with a conclusion in Chapter 6 summarizing the key results of the presented work and giving an outlook how the NNGA framework can be extended and improved.

A setup guide and a user guide can be found in the appendix.

2. Concepts

In this chapter we will give an overview of the concepts of neuronal networks and genetic algorithms which will later be needed to implement the framework.

2.1. Neuronal Networks

This section will provide an introduction in artificial neurons and neuronal networks. This knowledge is essential to be able to modify the program in a proper way. More detailed information on the features of the framework will be given in the next chapter.

2.1.1. Artificial Neuron

In consideration of the fact that artificial neurons are inspired from biologic neurons it is a good idea to take a closer look on the biologic version first. Figure 2.1 shows a neuron cell of a vertebrate.

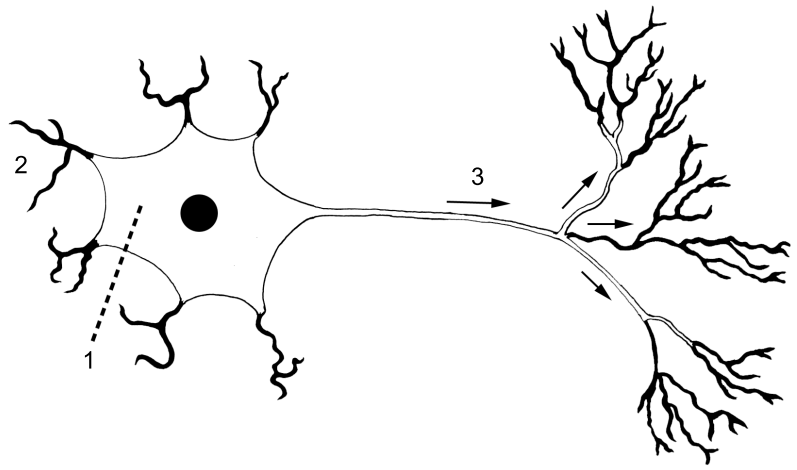

Figure 2.1.: Neuron cell: (1) is the cell body, (2) are dendrites leading to the cell and (3) is the axon leading output signal to other neurons.

Now we need to develop a mathematical model to be able to calculate like a neuron cell. We can describe a neuron as an n-ary function which maps n

inputs to one output. $x_1 \times x_2 \times \ldots \times x_n \mapsto o$ where $x_1, \ldots, x_n, o \in \mathbb{R}$. Usually an activation or transfer function φ is applied to the output. This function for a given neuron k can be defined as following:

$$a_k = \varphi(\sum_{j=0}^{n} w_{kj} x_k)$$

Where a_k is the output of the activation function, w_{kj} is the weight of the input from node j for the node k, x_k is the input from node k and n is the number of neurons. The output of a single neuron can be the input for other neurons or build the output of the network.

In order to make this model more flexible we introduce a bias value. The introduction of the bias changes the formula slightly:

$$a_k = \varphi(b_k + \sum_{j=0}^{n} w_{kj} x_k)$$

Here b_k determines the bias for a neuron k. The bias can also be interpreted as a neuron with constant output. With this we have found a quite appropriate model for a biologic neuron. The next step is to connect them in order be able to perform calculations.

2.1.2. Neuronal Network

With this model of a neuron we have a powerful tool to perform calculations. To make more difficult calculations possible it is necessary to build a *network* of single neurons. In the current implementation fully connected neuronal networks are being used. This means that each single neuron is connected to every other neuron directly. If we consider the neurons as nodes and their connections as arcs, a completely connected neuronal network can be seen as a complete directed graph. For many purposes some of the connections can be removed to reduce the calculation time. This can easily done in the framework by changing a few methods. We will discuss the required changes later in detail.

Having build up an artificial neuronal network we want this network to calculate values in a previously specified way. This is a quite hard job if we consider games like tic-tac-toe or our variant of a capture-the-flag game. The use of genetic algorithms makes this customization of the neuronal network more feasible.

2.2. Genetic Algorithms

Genetic algorithms are a technique for finding approximate solutions for various problems. Most of those problems are in the category of search or optimization problems. The genetic algorithms are a subclass of evolutionary algorithms. The idea for evolutionary algorithms which try to resemble evolution in nature comes from biology. It is obvious that those algorithms cannot outreach specific algorithms but what they can do is to provide a good approximation. In many real-world problems a good approximation is sufficient.

But what are the special characteristics of a genetic algorithm? And where is the connection between neuronal networks and genetic algorithms?

Genetic algorithms use well-known concepts such as inheritance, mutation, crossover and selection. A genetic algorithm works on a *population* of individuals and generates starting from an initial generation G_0 one or more descendant-generations G_n where $n > 0$. In practice the initial generation is a population of individuals initialized with random values. To simulate selection the algorithm builds up a ranking using a so called *fitness function* and replaces the rather worse individuals by new generated individuals. The choice of an appropriate fitness function is crucial. An inappropriate fitness function would lead the evolution in a wrong direction. Aside from the fitness function some other parameters for the genetic algorithm are also important. Those parameters define, e.g., which percentage of the population should be replaced or how often a crossover should happen. There is of course a quite big number of parameters which can be used to "customize" the genetic algorithm.

Irrespectively of the parameters other customizations are possible: It is for instance also possible to enable a crossover between the individuals of different populations. For more information on genetic algorithms beyond that what is discussed here we want to refer to [Fog00].

Now we will consider the ideas and problems of the genetic algorithm which is used in the framework. A more detailed description can be found in [EK07] (section 5).

First we have the concept of *elite selection*. This means that we select the networks with the best score. They will "survive" until the next iteration.

Then, with *random selection* some randomly selected, quite good networks (according to the score) will "survive" too. This is important because otherwise the diversity of the population would decrease to fast.

Mutation, which causes rather small changes, is applied to some of the networks to generate slightly changed networks. If the change was advantageous the network will stay in the population, if not it is likely that it will be re-

moved. The networks which will be mutated are randomly selected from the whole population.

Random Creation "generates" new networks with random configurations. Those new networks will replace the networks having a poor score in the next iteration.

Moreover, the idea of *Crossover* is very important but not easy to handle. The basic idea is to have several populations which evolve independently. After some time offsprings, which have qualities of individuals from (at least) two different populations, are created. Another idea is the intra population crossover: Here some parts of the networks are exchanged within a single population. The choice of the frequency of the inter population crossover is a crucial setting. A good setting is very important for good results. An idea to overcome the problem of finding optimal settings is the concept of metaevolution. We will discuss this in the next section.

The interested reader is referred to [EK07] where those ideas are discussed in detail and where the concepts (implemented in NNGA were applied to build a control system for a mobile robot.

Equipped with those powerful "tools" we can take a closer look at the features and the implementation of the NNGA framework.

3. Design Approach

Now after we have dealt with the concepts which were relevant for the NNGA framework, we will consider some design ideas which provide a basis for some possible extensions of the framework. (Those features were not fully implemented until the release of this work.) However, some ideas of the distributed evaluation (via RMI) were used in the interface to the robot simulator used in [EK07].

In this section we will deal with some interesting concepts which can be useful when working with neuronal networks and genetic algorithms. First we will illustrate the idea of meta-evoluation which helps finding optimal parameters for the Genetic Algorithm (GA). Then analysis of genetic variance will be presented. Afterwards we will deal with parallel processing of fitness evaluation.

3.1. Meta-Evolution

This concept defines an optimization of the parameters of the genetic algorithm like the mutation rate, the percentages of individuals to be kept/mutated/rearranged through a cross-over, etc. Optimal values for these parameters are quite hard to find in general and depend often on the respective problem. Meta-evolution uses the information gained by the evaluation phase to optimize the parameters of the genetic algorithm. As the evaluation data are also needed by the genetic algorithm this concept needs just a little extra effort.

Each population has an independent parameter set that is evaluated by the change of the overall fitness function of the population's individuals over several iterations. The overall fitness function of the population's individuals will be calculated as the average of the top 25% individuals. The evaluation is done for the interval between two inter-population cross-overs by making a regression analysis of the overall fitness function and extrapolating the result by one inter-population cross-over interval into the future. Figure 3.1 shows an example of an evaluation of three populations. The dotted lines mark the estimated development of the population.

After evaluation, the configuration of the worst performing population will be overwritten by a copy of the configuration of the best performing population.

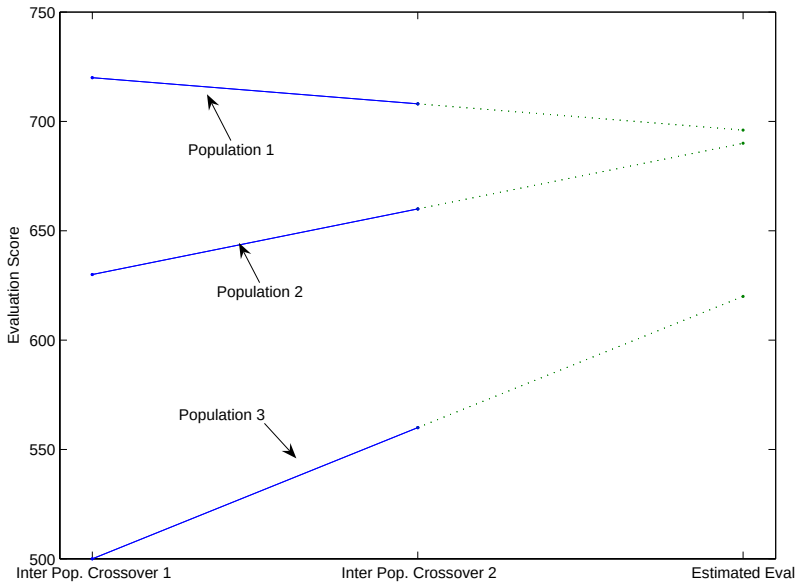

Figure 3.1.: Extrapolation of the evaluation result

Then the parameters of all but the best performing population are modified using a random mutation (also the copied configuration will be mutated).

Then, after the inter-population cross-over, the evaluation cycle restarts.

3.2. Analysis of Genetic Variance Among Individuals and Populations

Our hypothesis is that the partitioning of individuals into several populations keeps a greater variety among the individuals then having a single large population. In order to prove this hypothesis it is necessary to keep track of the variance among individuals and the variance between individuals of different populations.

Let $\sigma_i^2[k]$ be the variance of the k-th parameter (e.g., weight of a neural connection) of the top 25% individuals in the population i. Then $\bar{\sigma}_i^2$ denotes the average variance within population i.

Furthermore, let $\sigma_{ij}^2[k]$ be the variance of the k-th parameter of the top 25% individuals in population i or k and $\bar{\sigma}_{ij}^2$ denote the average variance within the two populations i and j.

Then, a value of $\frac{\bar{\sigma}_{ij}^2}{\sqrt{\bar{\sigma}_i^2 \cdot \bar{\sigma}_j^2}}$ denotes the correlation between the two populations i and j.

3.3. Parallel Processing of Fitness Value Evaluation

Many problems, e.g., the simulation of a mobile robot's behavior in a virtual environment, require considerable computation time in order to derive a fitness value for a given neural network. Each simulation consists of several steps and in order to get a dependable value it is necessary to repeat an experiment several times with different environmental starting conditions.

An example for a very time-consuming fitness evaluation is the simulation of a virtual environment with a mobile robot controlled by a neural network behavior. Even when using a faster-than-real-time simulation, the actual calculation time for a fitness function is in the order of seconds. Considering several populations with overall 100 individuals, the time for a single iteration is several hundreds of seconds, not to think of the time for 1000 iterations. By utilizing multiple computers and CPUs, this time can be easily cut down because the evaluation of the individuals within one iteration are independent and can be fully parallelized.

We assume that we have available a pool of PCs which are connected to the Internet. The average processing speed is assumed to be different, as it is the case with the operating system and the number of CPUs. Moreover, some of the PCs may be only available for a limited time, e.g., during night. These computers will run a fitness evaluation server that is able to calculate the fitness value for a given individual on request.

Our approach for parallel processing involves a main application that provides a user interface and performs the genetic optimization (mutation, crossovers, ...) over all individuals. However, for evaluation, an individual's configuration, i.e., the weights of the neural network, are transferred to one of the currently available fitness evaluation servers. After the fitness evaluation server has been instructed, the configuration of another individual is send to the next currently available fitness evaluation server. After evaluation, the fitness evaluation server returns the result and is given a new task. After having collected all the necessary fitness functions for one iteration, the genetic algorithm is applied and the next iteration is initiated.

If a server takes considerably long to report its result, the server is probed for a life sign message. If there is no life sign message it is assumed that the server has been shut down and the evaluation task is re-issued to another available server. The computer running the main application may also run a local server in order to utilize its processing power in a more efficient way. Multicore or hyper-threading systems may run several servers in order to optimize utilization.

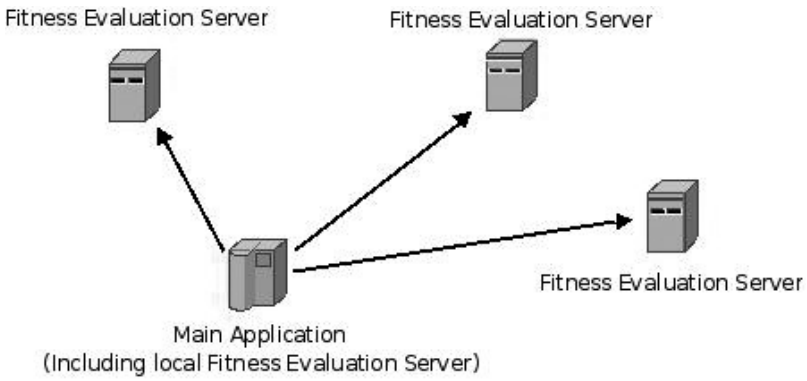

Figure 3.2.: Main Application and Fitness Evaluation Servers

In figure 3.2 the scheme of the fitness evaluation is shown.

Of course this strategy only makes sense if the evaluation function is difficult to calculate (and if the whole evolution procedure is intended to run over longer period). In this case the overhead for the distribution of the evaluations pays off.

4. Implementation

This chapter deals with some important implementation details of the NNGA framework. The information given in this chapter might be useful for those who plan to extend the framework.

The NNGA framework is implemented in Java. Therefore a Java Development Kit (JDK) compatible to version 1.5.0 is needed.

Some important features are:

- Persistence via XML
- Merging networks from other populations into others
- Basic RMI capabilities
- Dynamic class-loading (reflection-API)
- Extensible configuration system
- Various visualization and export functions

4.1. Persistence

Persistence belongs to the key-features as it is very important to store some states of one population. This feature comes out to be essential when tuning so parameters. It is also possible to save the whole system state or to define a interval for automatic backups. It has shown that those automatic backups are very useful when the software is performing longer calculations.

4.2. Merging

Sometimes it is a good idea to merge some networks of different populations to achieve a faster "development" of the networks. This is also possible with NNGA. Networks or whole populations can be imported into others. When importing networks into a population only the fittest networks will survive the next iteration (according to the selection of the GA). In NNGA this feature and the persistence system are closely connected.

4.3. RMI capabilities

During the development a request for RMI access occurred. It has shown to be a useful feature as well. With those (at the moment basic) RMI capabilities e.g. a robot simulator can provide the simulation feedback used in the GA to determine the current fitness. By using RMI it is not necessary that robot simulator and Main System are executed within the same Java Virtual Machine (JVM). As long as they are connected via a local area network or the internet they can interoperate in almost the same way. Even without knowing RMI very well it should be possible to write programs which communicate with the master system via RMI.

4.4. Dynamic Class-Loading

If the problem changes, which is used to train a network (or the idea of a slightly modified problem appears), it is very likely that the user will implement another problem class. With the dynamic class-loading it is possible to use classes which were just compiled without the need for restarting the whole system. So it is only necessary to put the fresh class into the classpath (so that it can be found by the JVM) and to set a configuration variable. In later time this feature will be available for networks, problems and many other important classes.

4.5. Extensible configuration system

This is one of the most important features. With help of the extensible configuration system it is possible to introduce set and modify different parameters used in the software. The configuration values are automatically stored in the state. So after adding no additional programming effort is needed. The new parameter can be used immediately (and is shown in the configuration Graphical User Interface (GUI) as well). Currently the following configuration value types are available.

- STRING
- INT
- LONG
- FLOAT
- BOOLEAN
- CINT (constant INT)

We want to point out that it is quite easily possible to extend this set of types by other desired types. The only condition is the availability of the new type in Java. Some examples concerning the usage of the configuration system will be presented in the userguide (section B.3).

Configuration entries can be identified via "URL"-like strings. Those string resemble internet domain entries - just the reverse direction. The whole configuration forms a configuration tree. The leaves of this tree are the configuration entries. E.g. *nn.nodes* denotes an (INT) entry which stands for the number of nodes in a single neuronal network.

4.6. Visualization and Export Functions

Another important topic are the export and visualization features. One possibility is to use some XML tools to extract the information from the saved state. But this might be too time-consuming to be used everytime when the user wants to know how the populations are developing. Therefore some visualization features generating development diagrams were implemented. In Section 5 we will show some Figures (Figures 5.1, 5.2 and 5.3) generated by this Visualization feature. (Those figures are screenshots which were taken from NNGA). It is also possible to export those fitness development data as Comma Separated Values (CSV) file. So it is possible to import the date easily into other tools or statistical software.

5. Results and Discussion

Although, NNGA is written in Java it is performing quite good on modern systems. In our experiments it has shown that after a few hundred iterations the fitness increase slows down. (Using some standard problems like the capture the flag game and fully connected networks) Therefore it is a good idea to merge two different networks after some hundred iterations. Then another iteration cycle will bring better results.

In table 5.1 a performance benchmark on an AMD Athlon 64 3500+ (2.21Ghz) with 1 gigabyte ram is shown. The problem was the capture the flag problem using 4 populations with 80 fully connected networks (with 6 nodes) and an inter-population crossover every 10 iterations. An average speed of roundly 188 iterations per minute was achieved, which is feasible for many applications.

Here we will describe the problem, which was used in the evaluation: The aim in the capture the flag game is to reach the middle of the field (i.e. the goal), performing as few steps as possible. The complicated part is that the player is being hunted by a guard. If the guard reaches the player, then the game is lost. In the evaluation, the player starts in the outer regions while the guard starts close to the goal. To improve the situation for the player, the guard is a little bit slower. (The player performs 4 steps while the guard can only perform 3 steps)

Iterations	Time needed (in seconds)	Iterations per minute
50	16,328	183,733
100	31,015	193,455
150	47,406	189,849
200	65,250	183,908
250	79,953	187,610

Table 5.1.: Performance Benchmark of NNGA

Now we take a look at the development of the best population (again, we consider the capture the flag problem with the above setting).

Figures 5.1, 5.2 and 5.3 show the fitness diagram of the best population after 50, 100 and 150 iterations. The scores of the best population is drawn in red, while the population average is drawn in blue. In this example it can be seen that the improvement of the performance slows down gradually.

It has to be mentioned that runtime of the evaluation function is the bigger part of the whole runtime. Moreover, the evaluation function is much more often executed that the genetic algorithm. Clearly the evaluation function is heavily determined by the problem which is being considered. This means that the choice of the problem (and in consequence the evaluation function) determines the overall performance in large parts. Therefore, it is quite difficult to measure the performance of the framework. Nevertheless, this example should give at least a feeling of the abilities of NNGA.

One way to increase the performance is to use other network types than fully connected networks or to tune the GA. Another possibility is searching for subnets of a single network which calculate a constant or a constant plus an offset. This subnetwork can be replaced by a single node which calculates this value. This will decrease the calculation time but also interfere with the GA. Therefore, this method only makes sense after the network has been brought to a reasonable performance level. There are - of course - many other possibilities to increase the performance. The last two examples shall motivate (and inspire) the interested reader to experiment with NNGA.

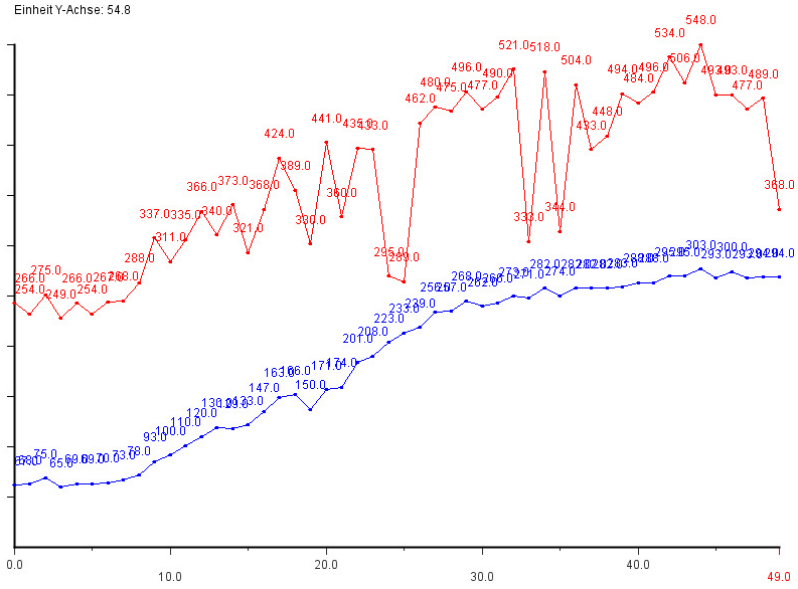

Figure 5.1.: The development of the best population within 50 iterations

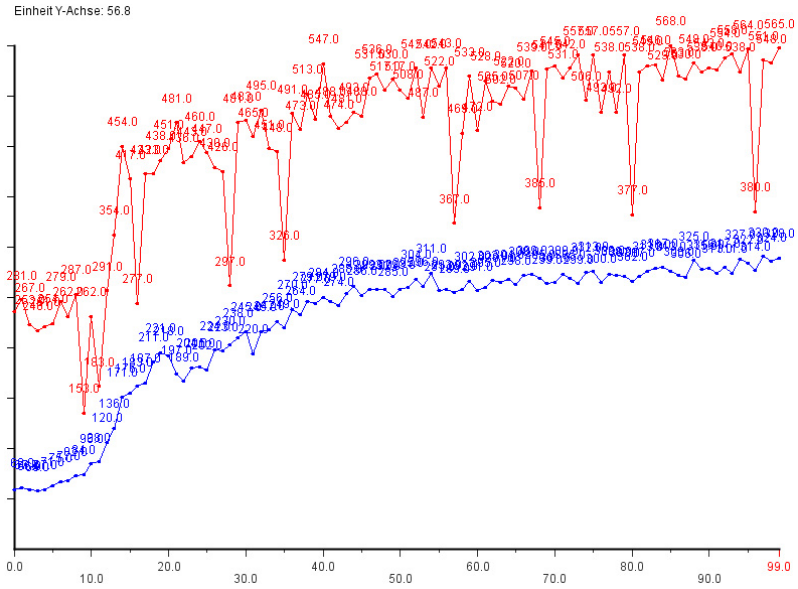

Figure 5.2.: The development of the best population within 100 iterations

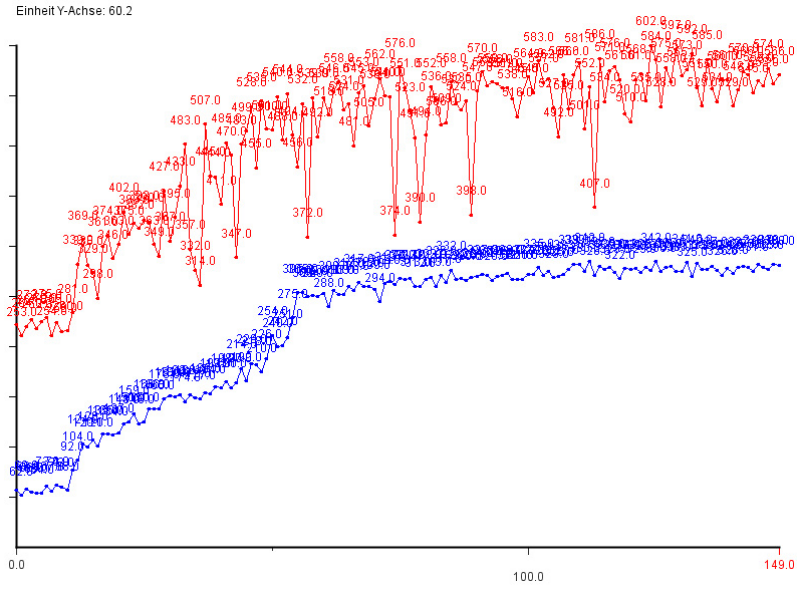

Figure 5.3.: The development of the best population within 150 iterations

6. Conclusion

We have dealt with the basic concepts of neuronal networks and genetic algorithms in order to provide a basis for the introduction of the NNGA framework. Some interesting concepts like meta evolution, variance among individuals and whole population and parallel processing of fitness evaluation were discussed. This work presented some ideas how those concepts could work together with NNGA.

Then we have seen implementation details and some interesting features and we have enumerated some interesting features which were not implemented until now. NNGA helps evaluating, testing and working with different types of neuronal networks and genetic algorithms. So it is not longer necessary that some general parts need to be re-implemented every time. A framework providing various features like persistence, visualization and distributed evaluation is ready to use. Anyway it is important to note that users of NNGA should know at least basics of programming in order to be able to extend it in a reasonable way.

Furthermore we have looked at the performance of the framework on currently used "standard" computers. It has turned out that NNGA is quite efficent and some possible improvements were discussed.

Still further development is possible and we hope that NNGA will be extended in the near future.

Bibliography

[EK07] W. Elmenreich and G. Klingler. Genetic evolution of a neural network for the autonomous control of a four-wheeled robot. In *Sixth Mexican International Conference on Artificial Intelligence (MICAI'07)*, Aguascalientes, Mexico, November 2007.

[Fog00] David B. Fogel. *Evolutionary Computation: Towards a New Philosophy of Machine Intelligence*. Wiley-IEEE Press, second edition, 2000.

[Heb49] Donald O. Hebb. *The Organization of Behavior*. Wiley, New York, 1949.

[MP43] W. S. McCulloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

A. Setup Guide

This guide will help the user to get a executable version of the nnga environment.
There are two ways to run the framework:

- Using the program without Eclipse
- Using the program with Eclipse

If the user wants to adopt the program it is recommended to use it with Eclipse.

A.1. System Requirements

The program requires for execution a Java 1.5 compatible environment. It has been tested with Sun Java 1.5.0_08 and Eclipse Java Compiler.

A.1.1. Obtaining the Sourcecode

For obtaining the source code it is the best idea to write an email to the supervisor or the author of this work. The current email addresses can be looked up using `http://whitepages.tuwien.ac.at`.

We can provide either a current snapshot of the program (including the sourcecode) or (if needed) svn access.

A.1.2. Required Software for usage without Eclipse

Those who don't want to use Eclipse need at least a Javacompiler. It's much easier if Ant is available. Ant is a make for Java Programs. It can be obtained from `http://ant.apache.org/`. With Ant installed the user just changes to to source root directory and types `ant` to compile the program. In the `build.xml` are various other targets. Those targets are comparable to the make targets. There are targets to build a jar archive with the compiled program inside and to build the javadoc documentation.

A.1.3. Required Software for usage with Eclipse

Obviously Eclipse is needed. It can be downloaded from `http://www.eclipse.org/`. Because Eclipse is a quite large program it's better to download it from a near mirror. For those who are in the area of the Vienna University of Technology the URL would be: `http://gd.tuwien.ac.at/softeng/eclipse/`

It is strongly recommended to use the subclipse plugin in combination with eclipse to make the svn-checkout more convenient.

If the user plans to build or change the GUI it's a good idea to install the *Visual Editor*. The Visual Editor is also an eclipse plugin which can easily installed. Just select in Eclipse: `Help > Software Updates > Find and Install`. Now the user selects `Search for new features to install` and after selecting the *Callisto* site the Download of *Visual Editor* is possible. Other plugins can be downloaded following this procedure as well.

A.1.4. Problems concerning the Heap Size

If the XML export function is used with large datasets it is possible that a *OutOfMemoryError* will occur. The GUI will prevent the application from terminating and just show the error message. The reason for this error is that the *jdom* library, which is used for the XML import/export as lower layer, uses the whole heap memory of the JVM. This results in a *OutOfMemoryError*. Fortunately this error can easily be avoided: The user just has to increase the heap memory size to a value around *256MB*. For the Sun JVM the parameters are: `-Xms64m -Xmx256m` *-Xms???* sets the amount of heap memory which is reserved during the startup of the JVM and *-Xmm???* sets upper limit for the size of the heap memory. Using those parameters a startup could look like `user@foo:~$/NNGASvn java -Xms64m -Xmx256m TestGUI`.

B. User Guide

This user guide explains how to get NNGA running and what startup parameters could be used (e.g. in scripts).

Moreover we provide a short introduction into the usage of the configuration system and explain how the sourcecode can be modified in order to add a new configuration parameter.

B.1. Starting the Program

Provided that the program was compiled and set-up correctly, it can be started by changing to the main source directory and typing the command `java TestGUI`. If the program was compressed to a `.jar` file it can be started by simply double-clicking on the file or by typing `java -jar ???` (where ??? stands for the name of the `.jar` file). If needed it is possible to append the arguments to this call. Those arguments will be passed to the started program. The currently implemented options are:

- `-f <filename>` This argument makes the program to load it's state from the given XML file at startup. This feature helps to simplify startup scripts.
- `-r <filename>` Given this argument the program loads it's state as described for the `-f <filename>` option. In addition it starts an unlimited calculation. This option can build the base for a watchdog script.

Of course it is also possible to use NNGA within eclipse. Sometimes it is very useful to use the debugger or the version control features of eclipse when working with NNGA. For questions concerning the use with eclipse we refer to the eclipse documentation (available on the eclipse project page or within the help menu of eclipse).

B.2. Main Menu

A few moments after starting up the program the main menu will appear. Figure B.1 shows how the main menu will look like.

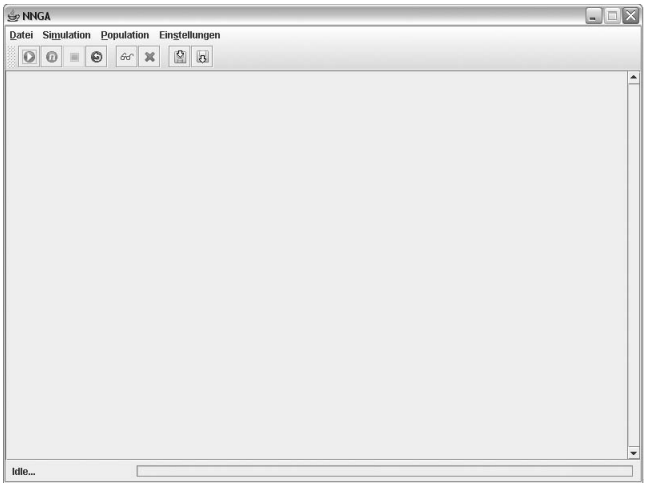

Figure B.1.: The main menu

All main features can be controlled from this menu.

B.3. Configuration system

The configuration system provides a central facility for different sorts of parameters. Persistence is provided for the parameters as well. Different parameter types reduce the risk of erroneous/unwanted changes. We will show how to introduce a new value and how this new value can be used in the framework.

All initial (non problem-related) parameters are stored within *NNGAConf* class in the variable `protected String[][] ivalues`. To add a new parameter e.g. "backup.fileextension" of type *STRING* it is only necessary to add a line {`"backup.fileextension","STRING",".bck"`}. Here *.bck* stands for the initial value of the new parameter "backup.fileextension".

Now the new parameter can already be used. To get the current value the following method call is needed: `conf.getValue("backup.fileextension")` (Provided that *conf* is a reference to the *NNGAConf* instance).

Setting a parameter to a new value works analogously: The method call `conf.setValue("backup.fileextension",".newbck")` sets the parameter to a new value.

If the NNGA state is stored or loaded, the system configuration is automatically stored or respectively loaded as well.

B.4. Starting the Calculation

There are two possibilities to start the calculation: With or without iteration limit. If an iteration limit is set then the calculation will stop as soon as the limit is reached. In the other case (i.e. Iteration without limit) the calculation will continue until a stop is requested by the user. It is recommended to enable the automatic backup system via the `backup.*` configuration variables. So almost no data will be lost if the program is interrupted while the calculation is running.

B.5. Export and Import

NNGA supports various import and export functions. First it should be mentioned that it is possible to export the development data of all populations as a CSV file. This can be useful if the user wants to analyze the data with some external program.

The other Export/Import functions use XML files. The first possibility is to export (or import) the whole program-state. This can be done by using the floppy-disk symbols in the toolbar. This action will export the program state and the configuration - i.e. it will run the XML export method in the *World* and in the *NNGAConf* class. This is especially useful for making snapshots of the current work. Note that the automatic backup uses exactly these functions.

The second possibility is to export only some of the populations. So it is possible to import previously saved populations into the system. Moreover, the user can import single networks from a previously saved population into an existing one. In this case the user must decide whether the population size should be increased or not. If the population size stays the same, the population will remove the worst ones of the surplus individuals after the next evaluation.

Note that no additional code change is needed if a new configuration parameter was introduced. The `NNGAConf` will export (import) every stored value.

It is recommended to set the history export configuration parameters to feasible values. Otherwise, the XML filesize could become very large.

B.6. Adding a new network type

Integrating a different network type into the system is quite easy (although some knowledge of programming and the code is required):

- First a new network class implementing the `INetwork` interface must be implemented. (The `CompleteNetwork` class could serve as a kind of "prototype" - Just copy the file and change what is needed.) Check whether the currently active problem algorithm supports this kind of network.
- Compile and test the new class. Then copy it into the NNGA source folders.
- Change the `nn.classname` configuration value to the complete classname (including packages!)
- Start NNGA.

In principle it should be possible to change the network class while the program is running. (However, it is required to reinitialize all populations and that the problem is compatible with the new introduced network class) Maybe some view modes are not applicable if they are not supported by the new network. If problems of this kind occur, check the network implementation.

B.7. Changing the considered problem

The steps required to change the problem are quite similar to those needed for changing the network class. However, this change is a little bit more difficult.

- First a new problem class implementing the `IProblem` interface must be implemented. Take a look at the `CTFProblem` class to get an idea how the methods could be implemented.
- Compile and test the new class. Then copy it into the NNGA source folders.
- Check whether the currently active networks work together with this new problem.
- Change the class-names and constructors in the `NNGAMainFrame` class to match the new problem.
- Start NNGA.

Of course a problem change is a quite major change. Clearly the problem evaluation visualization (i.e. replay) will very likely not work. The most important changes will be in `evaluateNet`, since this method controls the evaluation

of the network configurations (and assigns the score). A possibility to change the problem without changing code is planned in future releases. The same holds for the problem evaluation visualization. At the time it is recommended to keep several branches of NNGA in the SVN repository. Every branch holds a kind of problem. Tuning or slightly adopting a problem is possible by using some new introduced configuration values. (Quite analogously to the various different crossover methods used in the GA)

Now the user is familiar with the basic operations of NNGA. This userguide was intended to overview the features. The other features - not explained here - are rather self explaining.