Masaryk University
Faculty of Informatics

Evolutionary artificial neural network

Master thesis

Bc. Ondrej Verbó

Brno, autumn 2013

Declaration

Advisor: RNDr. Tomáš Brázdil, Ph.D.

Acknowledgement

I’d like to thanks to my advisor, RNDr. Tomáš Brázdil, Ph.D., for his guidance and patience, my family and my friends for their support during the studies.

Abstract

Thesis collects information about the evolutionary artificial neural networks (EANNs). EANNs refer to an area of an artificial neural network (ANN) used together with an evolutionary algorithm (EA). The evolutionary algorithm can be applied to different stages and parts of the artificial neural network's life cycle. Some of the collected information is implemented. The implementation uses artificial neural network framework called Neuroph which implements common ANNs types and their learning algorithms. Thesis compares the results between ANN and EANN on known problems and also experiments with real estate data. The network in our experiments is used to predict flat prices according to given set of attributes.

Keywords

prediction, evolutionary algorithms, genetic algorithms, artificial neural networks, evolutionary artificial neural networks

Contents

1 **Introduction** 1
2 **Artificial Neural Networks** 3
 2.1 Neuron 3
 2.2 Artificial Neural Network 4
  2.2.1 Architecture 4
  2.2.2 Dynamic 5
3 **Evolutionary Algorithms** 9
 3.1 Components 9
 3.2 Evolution Process 10
 3.3 GA, EP, ES 12
4 **Evolutionary Artificial Neural Networks** 14
 4.1 Design the Architecture 14
  4.1.1 Direct Encoding of the Topology 15
  4.1.2 Indirect Encoding of the Topology 18
   Parametric Representation 18
   Developmental Rules 19
  4.1.3 Evolving Transfer Functions 22
 4.2 Evolution of the Configuration 23
  4.2.1 Binary Representation 24
  4.2.2 Real Number Representation 24
 4.3 Simultaneous Evolution of the Topology and Configuration 25
 4.4 Evolution of Learning Rules 26
 4.5 Feature Selection 27
 4.6 Use of the Generation 28
5 **Neuroph** 30
 5.1 MultiLayerPerceptron 30
 5.2 BackPropagation 31
 5.3 DataSet 31
6 **EANN Framework** 32
 6.1 jEvolution 32
  6.1.1 Components 32
  6.1.2 Evolution Process 33
  6.1.3 EvolutionaryAlgorithm 34
 6.2 jNeuroph 35

6.2.1 IBuilder, IExtractor, ITransformer 35
Limitations 36
6.2.2 NeurophApplication 37
7 **Experiments** 39
7.1 Procedure 39
7.2 Computer Hardware Data Set 40
7.3 Flat Prices 44
8 **Discussion and Future Work** 47
8.1 Discussion 47
8.2 Future Work 47
9 **Conclusion** 49
A **Class Diagrams** 50

1 Introduction

Evolution as the change in characteristics of populations over the generations is one of the most important processes on Earth. Thanks to the evolution were evolved complex structures, such as neural systems. Neural system coordinates the voluntary and involuntary actions of the body and transmits signals between its parts.

In computer science is the evolution process represented by the evolutionary algorithm (EA). To imitate the neural system were developed different types of artificial neural networks (ANNs). The EA performs different tasks over ANN, such as architecture design, configuration adaptation, learning rules adaptation, data set feature selection, or their combinations.

This thesis is organized as follows. Chapter 2 is a quick overview of Artificial Neural Networks. Each ANN consists of interconnected set of elements called neurons. The chapter focuses on one of the most common ANN which is MultiLayer Perceptron (MLP). It also describes the most commonly used learning algorithm called Back-Propagation.

The next chapter 3 is devoted to an overview of evolutionary algorithms. It describes the basic components and principles of EAs. It also discusses the differences between the three types of EAs, such as genetic algorithms, evolutionary programming and evolutionary strategies.

Evolutionary algorithms and artificial neural networks used together are discussed in detail in chapter 4. The aim of this chapter is to describe the four main techniques, such as architecture design, evolution of the network's configuration, evolution of the learning rules and input feature selection. It also discusses the combination of two of the mentioned techniques, such as simultaneous evolution of the network's architecture and its configuration. At last it describes how can be used an ensemble of the population to get the output information. In this chapter we also encounter some problems connected to EANN, such as many-to-one and one-to-many mappings and we discuss how we can avoid them.

Chapter 6 describes our EANN framework implementation. We describe the artificial neural network framework called Neuroph. It provides the ANN functionality to our EANN framework. We also describe our implementation in detail. As part of the framework were implemented the architecture design, evolution of the configuration and the simultaneous use of both of them for feed-forward artificial neural networks types.

The next chapter 7 compares our solution with the standard learning process of the ANN. To do this comparisons are used Neuroph framework and our implementation. The comparison uses the data set from UCI repository called *Computer Hardware Data Set* and another data set was provided by the company ReeGo Development s.r.o. The computer hardware data set is intended to predict the relative CPU performance and the second data set to guess the price of the flat. Both of the data sets are similar in number of attributes but their character is different. Also the number of instances per data sets is different.

Chapter 8 discusses the cons and pros of the EANN in general and proposes the future work. Chapter 9 concludes the results.

2 Artificial Neural Networks

Computational models inspired by neural systems that are capable of machine learning and pattern recognition are called artificial neural networks.

This chapter is organized as follows. First section describes the most elementary unit of each artificial neural network, neuron. The next section is devoted to artificial neural networks. It explains what is the architecture of the network and describes the network's dynamic. The chapter is based on [23, 2, 17, 8, 6, 13, 16].

2.1 Neuron

An artificial (or formal) neuron is a computational model based on a biological (or neurophysiologic) neuron. The formal neuron consists of: an input vector $\mathbf{x}$, a weight vector $\mathbf{w}$, an inner potential ξ, a transfer function σ and an output y (see figure 2.1).

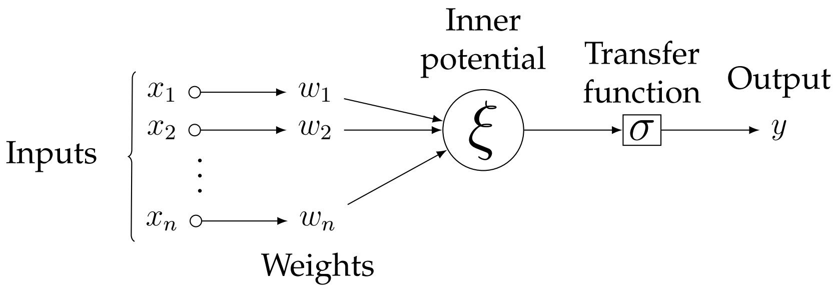


Figure 2.1: Formal Neuron

The input vector $\mathbf{x}$ consists of n real valued inputs. To each input corresponds one synaptic weight from weight vector $\mathbf{w}$. From x and $\mathbf{w}$ is calculated the inner potential ξ as follows:

$$\xi = \sum_{i=1}^{n} w_i x_i \tag{2.1}$$

From ξ is calculated the output value y using the transfer function $sigma$ as follows:

$$y = \sigma(\xi) \tag{2.2}$$

There are several types of transfer functions. Most commonly used are a sigmoidal (equation 2.3) and Gaussian (equation 2.4).

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}} \tag{2.3}$$

$$\sigma(\xi) = e^{\frac{-\xi^2}{2*\sigma^2}} \tag{2.4}$$

Described form of formal neuron is also called perceptron.

2.2 Artificial Neural Network

An artificial neural network can be defined as a directed weighted graph G where each vertex v of G represents exactly one formal neuron. The oriented weighted edge between two nodes is called a **connection**. The connection's value represents a synaptic weight w_{ji} from the neuron v_j to the neuron v_i.

2.2.1 Architecture

In order to define the network's architecture we have to define a topology and transfer functions.

Depending on the structure of graph G we recognize different types of *topologies*. In our work we concentrate on a special type of acyclic networks (graphs) called **feedforward** networks. Typical representative of the feedforward network is **Multilayer Perceptron** (MLP).

The feedforward networks have neurons divided into n disjoint sets called **layers**. Layers are organized in ascending order. First layer l_1 is called an **input layer**. It is mapped to the input vector $\mathbf{x}$. Following layers $l_i, 1 < i < n$ are called **hidden layers**. Number of hidden layers can vary. There is no specific rule to determine the correct number of them. Also the number of neurons in individual layers can be different. The last layer in the network is called an **output layer**. The value of the output layer is the output of the network and forms output vector $\mathbf{y}$.

Interconnections in multilayer perceptron network are organized as follows. There are no connections between neurons in the same layer. The output of each neuron from layer l_i is connected to neurons in layers l_j where $1 \leq i < j \leq n$.

We can see the feedforward topology of the artificial neural network with five inputs, one hidden layer with three neurons and output layer with one neuron on figure 2.2.

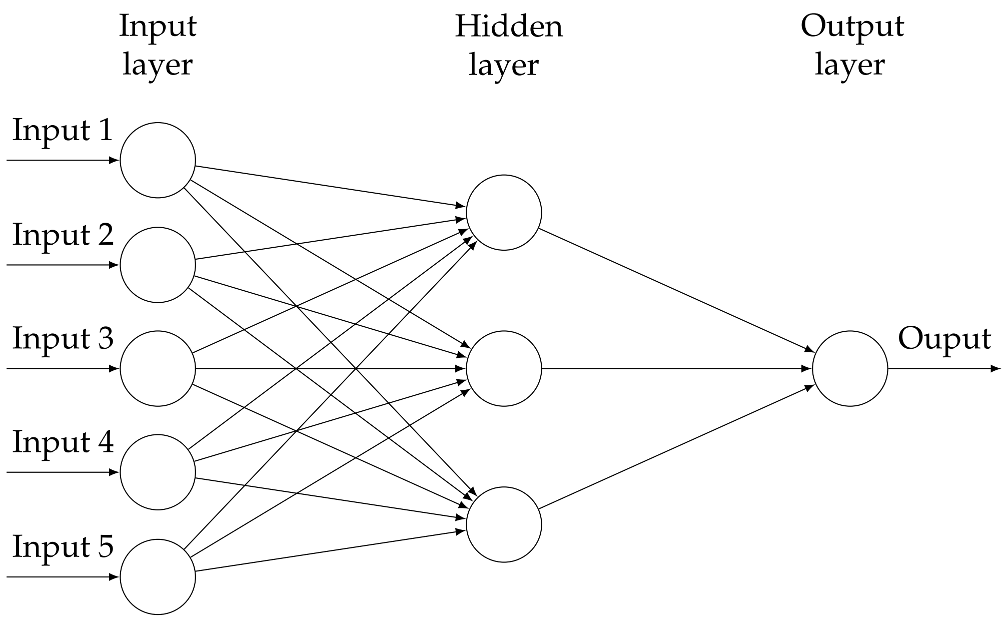


Figure 2.2: MultiLayer Perceptron

The common approach is to have only one type of the *transfer function* for all neurons in the network. But in some cases can be used multiple types of the transfer function. We discuss this possibility in section 4.1.3.

2.2.2 Dynamic

Artificial neural network performs learning over a data. Multilayer perceptron network utilizes a **supervised learning** algorithm called **backpropagation**.

A supervised learning data is a set of values called a **data set**. Each item of the data set is a pair of values ($\mathbf{x}_i$, $\mathbf{d}_i$). $\mathbf{x}_i$ is the i_{th} input vector sent to the network and $\mathbf{d}_i$ is the corresponding desired output vector.

A supervised learning algorithm, such as the backpropagation, describes a function $f : \mathbb{R}^n \to \mathbb{R}^m$. The function f is given by an **active dynamic** and strictly depends on the network's topology and its **configuration**. The configuration is a set of all synaptic weights in the network.

Lets consider the MLP network with static topology and each connection in the network having a synaptic weight. Neurons in the input layer are set to the input of the network. The rest of neurons are left in the initial state. The initial state means their output values are set to 1. Then are calculated output values for each layer $l_i; 1 < i \le n$ in ascending order as follows:

1. output values from the layer l_{i-1} serve as input to the layer l_i
2. calculate the inner potentials for each neuron in layer l_i
3. calculate the output values in for each neuron in layer l_i

An advantage of artificial neural networks is that they can adapt their synaptic weights according to the prescribed input / output behavior. This adaptation is described by an **adaptive dynamic**. The adaptive dynamic serves to learn how to describe the function f. This process is also called a **learning**.

The backpropagation learning algorithm uses an error of the configuration $E(\mathbf{w})$ to adapt synaptic weights. Lets assume the data set $\mathcal{T} = \{\mathbf{x}_k, \mathbf{d}_k | k = 1, .., p\}$, where p is the number of the data set's items. For the data set $\mathcal{T}$, the configuration $\mathbf{w}$ and the output vector $\mathbf{y}$ we define the error $E(\mathbf{w})$ as follows:

$$E(\mathbf{w}) = \sum_{k=1}^{p} E_k(\mathbf{w}) \tag{2.5}$$

$$E_k(\mathbf{w}) = \frac{1}{2} \sum_{j \in \mathbf{y}} (y_j(\mathbf{w}, \mathbf{x}_k) - d_{kj})^2 \tag{2.6}$$

At the beginning of the learning process are all synaptic weights initialized to random values near zero or they are set according to some prior knowledge. The algorithm calculates the sequence of synaptic weights vectors $\mathbf{w}^{(0)}$, $\mathbf{w}^{(1)}$, etc. For the step t the synaptic weights vector $\mathbf{w}^{(t)}$ is calculated as follows:

$$w_{ji}^{(t)} = w_{ji}^{(t-1)} + \Delta w_{ji}^{(t)} \tag{2.7}$$

Where $\Delta w^{(t)}$ equals to the negative gradient of the error function (equation 2.8).

$$\Delta w_{ji}^{(t)} = -\varepsilon \frac{\partial E_j}{\partial w_{ji}} (w^{(t-1)}) \tag{2.8}$$

In equation 2.8 $\varepsilon(t)$ is a **speed of learning** in step t. The speed of learning influences the quality and speed of learning and it is also called **learning rate**.

Described approach belongs to bigger family called **gradient based algorithms**. They use a **gradient information** to determine the direction of the steepest ascent of the error function and always reach local optima. But local optima does not have to necessarily represent the best solution. It means we do not have to find the best configuration of the network. There is also no guaranteed way how to determine whether the reached optima is local or global.

In order to calculate $\Delta w_{ji}^{(t)}$ we have to calculate the gradient of the error function from 2.8. To do so we transform the gradient into the sum of gradients of partial error functions:

$$\frac{\partial E_j}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}} \tag{2.9}$$

Using the rule of composite function we get:

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \frac{\partial y_j}{\partial \xi_j} \frac{\partial \xi_j}{\partial w_{ji}} \tag{2.10}$$

By the differentiation of the inner potential from 2.1 we get $\frac{\partial \xi_j}{\partial w_{ji}}$ as follows:

$$y_i = \frac{\partial \xi_j}{\partial w_{ji}} \tag{2.11}$$

By the differentiation of the transfer function from 2.3 we get $\frac{\partial y_j}{\partial \xi_j}$ as follows:

$$\begin{aligned}\lambda_j y_j(1-y_j) &= \frac{\lambda_j}{1+e^{-\lambda_j xi_j}}(1-\frac{1}{1+e^{-\lambda_j xi_j}}) = \\ &= \frac{\lambda_j e^{-\lambda_j xi_j}}{(1+e^{-\lambda_j xi_j})^2} = \\ &= \frac{\partial y_j}{\partial \xi_j} \end{aligned} \tag{2.12}$$

We get $\frac{\partial E_k}{\partial y_j}$ as follows:

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \tag{2.13}$$

in case if $j \in \mathbf{y}$ is the output neuron. That corresponds to the error of neuron j for k_{th} item from $\mathcal{T}$.

To calculate the error for the neuron in hidden layer we amend $\frac{\partial E_k}{\partial y_j}$ as follows:

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j} \frac{\partial E_k}{\partial y_r}\frac{\partial y_r}{\partial \xi_r}\frac{\partial \xi_r}{\partial y_j} = \sum_{r \in j} \frac{\partial E_k}{\partial y_r}\lambda_r y_r(1-y_r)w_{rj} \tag{2.14}$$

where j is a neuron from the hidden layer. In order to calculate the partial derivative of the neuron j we follow the order from the output layer to the first hidden layer. We can compute the partial derivative for the neuron j in case all neurons which have the input connection leading from j have the partial derivative already computed.

More information about the gradient and backpropagation and other learning algorithms we can find in [13, 16].

3 Evolutionary Algorithms

Evolutionary algorithms (EAs) are based upon evolutionary principles of the natural evolution, such as a selection, a recombination and mutation and a reproduction. It follows the principles of *survival of the fittest*.

Currently we recognize three main types of EAs: genetic algorithm (GA), evolutionary programing (EP) and evolution strategy (ES).

This chapter is organized as follows. First section discusses the evolution and evolution process. It defines the basic terminology and common principles of the EA. The next section describes the differences between individual types of EA as described above. Chapter is based on [23, 19, 4, 6, 21].

3.1 Components

Each evolutionary algorithm has a **population**. The population is a set of n independent elements called **individuals** or **chromosomes**. Each chromosome consists of another m independent elements called **genes**.

The chromosome's main purpose is to represent a possible solution for some problem we want to solve. The possible solution is held in a form of an information encoded into genes.

A set of all genes in a chromosome is called a **genotype**. A feature resulting from the genotype is called a **phenotype**.

Evolutionary algorithms are often used to solve optimization problems. The main purpose of the EA is to find the near optimal solution. To determine how good is the possible solution [1] serves a **fitness function**. The fitness function determines how the solution is evaluated by the algorithm and how is the problem we want to solve defined. If the value of the fitness function reaches some predefined value, the solution is marked as near optimal.

In order to construct the evolutionary algorithm, we have to define the problem we want to solve and how it will be represented by

1. the distance from the near optimal solution

the fitness function. Then we need to determine the chromosomes and genes structure and representation.

For example we want to find the satisfying configuration of the artificial neural network. Satisfying configuration is found when the error of the network's configuration is not bigger than 0.001. The population consists of a set of configurations. The chromosome is represented by a set of synaptic weights of one network. The gene is the binary value of one synaptic weight of the given network. The fitness function is the value of the error of the network. Genotype is the set of all distinct values of the synaptic weights within one chromosome. The phenotype is the error of the given network.

3.2 Evolution Process

The purpose of the evolution process is to find the near optimal solution. In order to find it the population changes. Chromosomes can exchange the information between them. This exchange results into new chromosomes. Another possibility is to slightly change the information within the chromosome. All chromosomes then compete each other. Newly created chromosomes can be added to the population and also existing chromosomes can be removed from it.

These changes happen in iterations. The iteration is often called an **epoch**. Each epoch is represented by the current population. The current population is often called a **generation**.

At the beginning of the evolution process is generated an **initial population**. In initial population each chromosome carries randomly assigned information about the problem to be solved. When the initial population is generated each chromosome is evaluated by its **fitness**. Fitness is the value calculated by the the fitness function.

Next step in the evolution process is to select a subset of chromosomes which will pass their information to the next generation. We call this subset **parents**. This process is called a **selection**.

The selection is followed by a **recombination process**. Its purpose is to create new chromosomes from parents by applying **recombination operators**. Recombination operators serve to exchange (recombine) the information between two parent chromosomes in a process called a **crossover** or to amend it what refers to **mutation**. Result of the recombination is a set of newly created chromosomes. It is called an **offspring**. Also each chromosome belonging to the offspring's set is evaluated by its fitness.

The crossover is a two way exchange of bits of informations from one chromosome to another. We can see an example of the crossover on figures 3.1a (pre single-point crossover chromosomes) and 3.1b (post single-point crossover chromosomes). We can find more details about the crossover in [23, 19].

01010101	01011010
10101010	10100101
(a) Pre crossover chromosomes	(b) Post crossover chromosomes

Figure 3.1: An example of single-point crossover recombination operator applied to two different chromosomes

The mutation represents the process where randomly chosen chromosome can change the information on randomly chosen gene. An example of the mutation is shown on figures 3.2a (pre mutation chromosome) and 3.2b (post mutation chromosome).

01010101	01011101
(a) Pre mutation chromosome	(b) Post mutation chromosomes

Figure 3.2: An example of mutation recombination operator applied to the chromosome

The last phase is called a **reproduction** (also often referred as a creation of the new population). From all chromosomes in both sets (generation and offspring) is created a new generation. There are three commonly known reproduction models:

- Generation model - original population is replaced by offspring.
- Steady state model - replace the chromosome with the worst fitness from the original population by the offspring with the best fitness.
- Overlaying model - replace only part of the chromosomes from original population by some of offspring (previous models are extreme cases of this model).

If after the reproduction a **stopping criteria** are not hit, the process continues with the selection process. Stopping criteria can be reaching some predefined number of iterations (epochs) or the near optimal solution is present in the generation.

The evolution process can be described by the following pseudo algorithm:

```
Result: near optimal solution
create the initial population;
evaluate each chromosome in the population by its fitness;
while not hitting the stopping criteria do
    select parents;
    recombine parents and create offsprings;
    evaluate each offspring by its fitness;
    reproduce the population;
end
```

Algorithm 1: Evolution process

In order to define the evolution process we have to determine which selection method will be used, how will be the recombination done and which reproduction model will be used.

3.3 GA, EP, ES

Genetic algorithms have in most cases binary genes and the chromosome is represented by a string composed from zeros and ones.

As the parent can be selected any of chromosomes in the current population. The result of the selection process are always pairs of chromosomes. Between them is the information exchanged using the crossover recombination operator. After that with a certain small amount of probability is applied a mutation operator. To create the new population for the next generation is used in most cases the generation model.

The representation of genes for the ***evolutionary programming*** can vary. It can be binary or problem specific. If the representation is problem specific, new special recombination operators have to be created. For example a mutation with the definition how is the chromosome mutated.

In selection process becomes every chromosome a parent. To create new chromosomes is used the mutation operator. It is applied to each parent and each parent generates exactly one offspring. The new population is created on elitists principle using the overlaying reproduction model. That means only the fittest individuals among the parents and newly created set of offspring are chosen to form the next generation.

Evolutionary strategies are in all ways similar to the EP. The gene's representation can be binary or more problem specific.

But the population of the evolutionary strategy contains only one chromosome. In order to create the offspring parent chromosome is mutated multiple times. For the mutation is in most cases used Gaussian mutation [2]. Then the fittest chromosome is selected as the new parent in the next generation.

Later research discovered that more than one chromosome in the population can lead to better results of evolutionary strategies. Although mutation still remains the only recombination operator. Also only the fittest chromosomes are chosen for the new generation.

2. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.183.2816&rep=rep1&type=pdf

4 Evolutionary Artificial Neural Networks

In previous chapters we have presented two different tasks of machine learning which fall under the scope of nature inspired artificial intelligence. In this chapter we focus on possibilities of their mutual use. This chapter is organized as follows.

First section discusses the possibility of using the evolutionary algorithm to design the architecture of the artificial neural network as we have described it in section 2.2.1. There was a huge research focusing on the evolution of the architecture in the past [21, 12, 9, 15, 11].

The next section is devoted to discussion of the evolution of the network's configuration. The learning process as we have described it in section 2.2.2 means to adapt synaptic weights. But this process can be replaced by the evolution [3, 1, 21, 12, 9, 14, 7].

Section 4.3 is an overview of the possibility of the mutual use of the evolution of the topology and configuration [3, 21, 9].

Evolution of learning rules is discussed in the next chapter. It focuses on the explanation of how can be the learning rule evolved and discusses the benefits of evolution of learning rules [3, 21].

Some data sets can be big what can lead to slow learning process of the artificial neural network. Restrict the size of the data set can lead to faster learning process. Also it can exclude the noise from the data set what can leads to better network's performance[7, 14, 10, 5, 3]. This possibility is discussed in section 4.5.

Last section of this chapter discusses an option how can be used an ensemble of chromosomes from the last generation to generate the output. Using an ensemble as output can lead to better generalization ability [24, 22, 21, 3, 20].

4.1 Design the Architecture

In section 2.2.2 we have stated: *"The function f is given by the active dynamic and **strictly depends on the network's topology** and its configuration"*. To adapt the configuration were designed multiple deterministic algorithms (see section 2.2.2). But there is no guaranteed way how to design the topology properly. It is the matter of an experience.

To help to solve this problem were used evolutionary algorithms. To evolve the topology we have to decide what exactly do we want to evolve. On that decision depends what kind of the evolutionary algorithm and which recombination operators will be used, what kind of information will be encoded into the chromosome and how its genes will be represented.

We can decide to evolve the whole topology of the ANN. In that case we are talking about a **direct encoding**. The other option is to evolve only some information about the topology, such as the number of neurons, the number of layers, connections between neurons etc. In that case we are talking about an **indirect encoding**.

Except of the topology, the architecture contains also another kind of information about the ANN. It is the transfer function of neurons. In section 2.2.1 we have mentioned: *"The common approach is to have only one type of the transfer function for all neurons in the network"*. As part of this section we will discuss the possibility of using using multiple transfer functions in the network and its evolution.

4.1.1 Direct Encoding of the Topology

The information contained in chromosomes holds complete architecture design, such as the number of layers, the number of neurons in layers and connections between neurons. It is also said that the information in direcrect encoding is maximal.

A chromosome is composed from n^2 binary genes where n determines the number of neurons used in the network. We can imagine the network as an $n \times n$ matrix M where the value on coordinate $[j, i]$ is considered to be a gene. The gene indicates the presence (1) or absence (0) of the synaptic weight w_{ji} between j_{th} and i_{th} neuron. The final chromosome is in the form of concatenated rows of matrix M.

For the network as on figure 4.1 the matrix M looks like as on figure 4.2 with the corresponding chromosome on the figure 4.3.

To evolve the architecture with the direct encoding are used genetic algorithms. The population usually consists of hundreds or thousands of chromosomes. In selection process are randomly chosen pairs of chromosomes to become parents. Using the crossover operator is derived the offspring. Then with certain amount of proba-

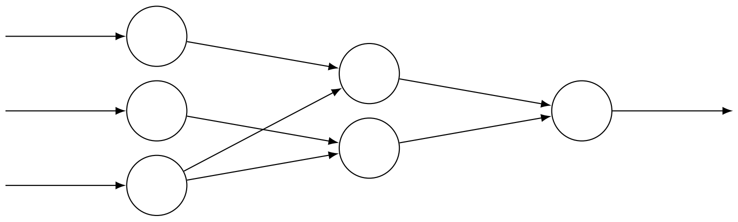


Figure 4.1: Artificial Neural Network

$$\begin{matrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Figure 4.2: Matrix M

000100000010000110000001000001000000

Figure 4.3: Chromosome

bility is randomly chosen one chromosome from the offspring which is mutated[1]. In the reproduction process any of reproduction models can be used.

When new generation is derived, each chromosome is trained by standard training process as described in chapter 2. After that is calculated its fitness using fitness function. The fitness function can evaluate the chromosome according to the size of the topology and we can look for the minimal or maximal topology. Other possibilities are to calculate the number of connections or neurons, determine the best performance (minimal error) or combinations of these options.

The advantage of the direct encoding representation for feedforward networks like MLP is that they can have connections only to higher layers. The bottom left half of the matrix is always filled with 0 so the final chromosome can be reduced to the half of its original size. We can see this reduction on figure 4.4. In general stands that

1. note that also no chromosome can be chosen

the implementation is very straightforward and easy.

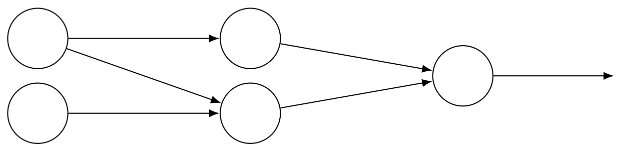
000100000100110001010

Figure 4.4: Reduced chromosome for feedforward networks

But this method suffers from couple of disadvantages. From the principles of EAs, size of the chromosome has to be constant. We have to decide what is the maximal number of neurons in the network. Also for bigger networks can be the evolution process slow as learning and evolution processes take more time to converge to near optimal results. The consequence is slow performance and huge computational time consumption.

Another problem with the direct encoding is connected to a **permutation problem**. The permutation problem refers to many-to-one mapping.

Imagine two isomorphic graphs (figure 4.5) which represent equivalent artificial neural networks. But these two networks will have different genes mapping (4.6) resulting in two different chromosomes (4.7). This results in crossover recombination operator being inefficient because it may result into a network isomorphic to the original ones. They will just have different mapping. This problem prolongs the evolution time and makes it inefficient [21, 3].


(a) Network A

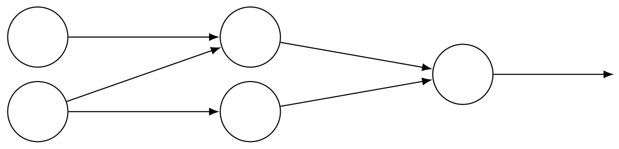

(b) Network B

Figure 4.5: Two Isomorphic Networks

$$\begin{array}{ccccc} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{array}$$

(a) Matrix M for the Network A

$$\begin{array}{ccccc} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{array}$$

(b) Matrix M for the Network B

Figure 4.6: Matrices of networks A and B

0011000010000010000100000

(a) Chromosome A

0010000110000010000100000

(b) Chromosome B

Figure 4.7: Chromosomes of networks A and B

This problem can be solved using evolutionary programming instead of genetic algorithms. The gene and chromosome representation remain the same but using only mutation to produce offsprings leads to prevent the permutation problem.

4.1.2 Indirect Encoding of the Topology

To discover the near optimal solution indirect encoding relies on the evolution more than the direct encoding.

The chromosome carries only partial information about the architecture. It can also contain some additional information about the learning algorithm used to train the ANN. As only partial information about the ANN is evolved, indirect encoding is often referred as minimal.

There are two main ways how genes in chromosomes can be represented, by **parameters** or **developmental rules**.

Parametric Representation

The parametric representation of indirect encoding is a straightforward way how to evolve the topology of the ANN. When using parametric representation we have to decide which parameters of the network we want to evolve.

We can decide to evolve the number of layers or neurons in the network or the number of connections between layers. We can also combine the possibilities. In that case are genes individual features. Also gene can be a specific information about the learning algorithm like the momentum or learning rate.

In the chromosome, the genes are usually encoded in binary form (so that the chromosome is a string of 0 and 1). The other possibility is to leave the genes in original, real number, form.

To evolve the topology can be used any kind of the presented EA. In this case the structure of the chromosome has to be predefined and static. Otherwise we would not be able to decode the information and rebuild the artificial neural network. The other possibility is to include the name of individual parameters into the chromosome.

Using the GA does not lead to the permutation problem. There is no way how to create two different chromosomes which could lead to the isomorphic decoded information.

If genes are encoded in original form recommended methods are EPs and ESs as GAs are designed to work with the binary representation.

Parametric representation suits mostly cases when we know what kind of topology we want to develop and which learning algorithm we want to use. For example if we want to develop feedforward network like MLP for BP learning algorithm.

Developmental Rules

Developmental rules are not so straightforward as parametric representation. At first we will describe the principle how they work and then how the evolution process works.

Developmental rules are similar to the formal grammar. We have a finite set of nonterminal symbols N, a set of sixteen terminal symbols Σ disjoint from N, a finite set of rules P and a starting symbol $S \in T$. It can be formally defined as a tuple (N, Σ, P, S).

The set of nonterminal symbols is mostly represented by capital letters from A to Z. The set of terminal symbols is in most cases represented by the lowercase alphabet from a to p. Each terminal symbol represents 2×2 unique binary matrix. We can see an example of such a matrix on figure 4.8.

Rules in P are in the form $n \to m$ where n is always the nonter-

$$\begin{array}{llll}
a \rightarrow \begin{matrix}0 & 0\\ 0 & 0\end{matrix} & e \rightarrow \begin{matrix}0 & 0\\ 0 & 1\end{matrix} & i \rightarrow \begin{matrix}0 & 1\\ 1 & 0\end{matrix} & m \rightarrow \begin{matrix}1 & 1\\ 0 & 1\end{matrix} \\
b \rightarrow \begin{matrix}1 & 0\\ 0 & 0\end{matrix} & f \rightarrow \begin{matrix}1 & 1\\ 0 & 0\end{matrix} & j \rightarrow \begin{matrix}0 & 1\\ 0 & 1\end{matrix} & n \rightarrow \begin{matrix}1 & 0\\ 1 & 1\end{matrix} \\
c \rightarrow \begin{matrix}0 & 1\\ 0 & 0\end{matrix} & g \rightarrow \begin{matrix}1 & 0\\ 1 & 0\end{matrix} & k \rightarrow \begin{matrix}0 & 0\\ 1 & 1\end{matrix} & o \rightarrow \begin{matrix}0 & 1\\ 1 & 1\end{matrix} \\
d \rightarrow \begin{matrix}0 & 0\\ 1 & 0\end{matrix} & h \rightarrow \begin{matrix}1 & 0\\ 0 & 1\end{matrix} & l \rightarrow \begin{matrix}1 & 1\\ 1 & 0\end{matrix} & p \rightarrow \begin{matrix}1 & 1\\ 1 & 1\end{matrix}
\end{array}$$

Figure 4.8: Binary matrix for a set of terminal symbols

minal and m is a 2×2 matrix composed of either terminals or nonterminals. Rules are repeatedly applied until only terminal symbols are left. Created matrix is then rewritten into binary matrix which represents the architecture of an artificial neural network.

For the better imagination we can assume the tuple (N, Σ, P, S) defined as on figure 4.9. We can see there five nonterminal symbols and sixteen terminal symbols with five rules and a starting symbol. The decomposition of this developmental rule is showed on figure 4.10a. Such a matrix is then rewritten into binary matrix using the binary matrix from figure 4.8. We can see the rewritten matrix on figure 4.10b. And from this matrix we can simply create the network which is showed on figure 4.11.

$G = (N, \Sigma, P, S)$
N = {S, A, B, C, D}
Σ = {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p}

$$P = \left\{ \begin{array}{lll}
S \rightarrow \begin{matrix}A & B\\ C & D\end{matrix} & A \rightarrow \begin{matrix}a & j\\ a & a\end{matrix} & B \rightarrow \begin{matrix}d & a\\ h & d\end{matrix} \\
 & C \rightarrow \begin{matrix}a & a\\ a & a\end{matrix} & D \rightarrow \begin{matrix}c & e\\ h & c\end{matrix}
\end{array} \right\}$$

S = S

Figure 4.9: Developmental rules definition

$$S \rightarrow \begin{matrix} A & B \\ C & D \end{matrix} \rightarrow \begin{matrix} a & j & d & a \\ a & a & h & d \\ a & a & c & e \\ a & a & a & c \end{matrix}$$

(a) Decomposition of the rules

$$\begin{matrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

(b) Coresponding binary matrix

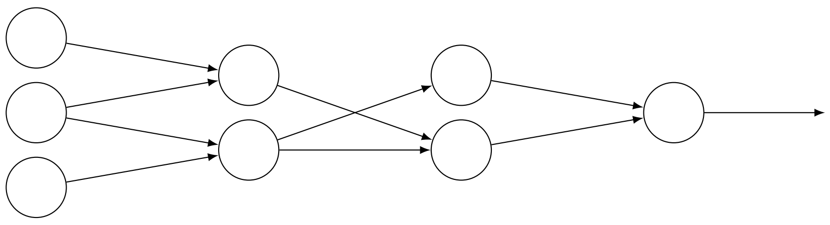


Figure 4.11: Decoded architecture of the network

The chromosome can be represented as the whole rule set or each rule can be standalone chromosome. Therefore before the evolution some rules have to be established (as per our example above). Depending on the number of nonterminal symbols used has to be chosen also the size of the chromosome. Also the position has to clearly determine if it is the start or the end of the symbol and also if it is left-hand or right-hand side of the rule. But this steps sometimes falls off as only left-hand side part of the rule is present in chromosome. Also mapping algorithms has to be used here.

To evolve developmental rules can be used any kind of EA but because of its complexity it is the EP or ES. Also there is no risk of permutation problem as evolved are rules, architecture is encoded indirectly.

4.1.3 Evolving Transfer Functions

Until now we have considered only topological organization as part of the architecture. But there is also another attribute which can play a role in evolving the architecture and that is the transfer function.

In chapter 2 we have mentioned sigmoid and Gaussian transfer functions which are considered to be the most commonly used. Also standard learning process uses only one type of transfer function for each neuron of the network.

Authors of [21, 11] proposed to use multiple types of transfer functions in one artificial neural network. They use sigmoid and Gaussian transfer functions to evolve the best ratio between them to ensure the best performance.

The gene in this case is a predefined code of transfer function used for given neuron. It can be encoded binary or by real value. Chromosome is then concatenation of all the transfer functions within the network.

Lets consider an artificial neural network with static topology as on figure 4.12a. Lets assume we want to use sigmoid and Gaussian transfer functions. Each neuron of the artificial neural network has exactly one transfer function. The sigmoid transfer function will have a code 1 and Gaussian 0. All neurons in the input and output layer have sigmoid transfer function. All neurons in the hidden layer have Gaussian transfer function. The chromosome then looks like on figure 4.12b.

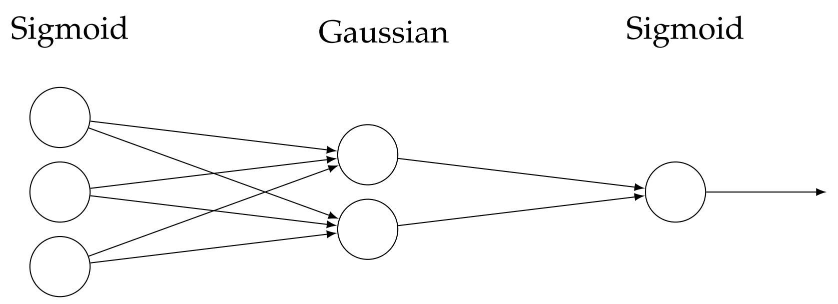
To evolve transfer functions can be used any of the EA. But using binary encoded chromosomes in association with the GA can lead to the permutation problem.

Another approach is to evolve directly the transfer function. In this case only one transfer function type is chosen but its attributes are evolved.

In case of Gaussian transfer function in form of equation 4.1 it can be constants a, b, c or d.

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}} + d \tag{4.1}$$

Genes are similar to genes of the parametric representation used to evolve architecture by the indirect encoding (see section 4.1.2). Also the evolution process is almost the same.



(a) Artificial Neural Network

111001

(b) Chromosome

Figure 4.12: Evolution of Transfer Functions

4.2 Evolution of the Configuration

In previous section we have outlined how can be evolved the architecture. Mentioned approaches rely on classical learning process of the ANN using the learning algorithm.This section describes how can be learning replaced by the evolution.

At the beginning of the evolution process we have to define the architecture. It remains static during the process.

Each synaptic weight represents one gene. The chromosome is then concatenation of all synaptic weights of the network as follows:

1. Take the first hidden layer $l_k; k = 2$, set it as current and select the first neuron $n_j; j = 1$.
2. Take every neuron n_i from the lower layer l_{k-1} and encode its connection to the neuron n_j as its synaptic weight w_{ji}. If there is no connection, encode connection as 0.
3. Repeat second step for each neuron n_j in the current layer l_k.
4. Move to the higher layer l_{k+1}, set it as current and select first neuron $n_j; j = 1$.
5. Repeat steps 2, 3 and 4 until output layer l_n is reached.

In order to encode the synaptic weight we have to make a decision what kind of gene representation we want to use. The representation can be **binary** or **real number**.

4.2.1 Binary Representation

In binary representation is every synaptic weight encoded into a binary string. But the binary representation has several risks. If we choose short length of the binary string, the network does not have to train properly as we can be loosing some information. On the other hand, if we choose the length too big, the network can easily overfit what leads to poor generalization ability. The length is also connected with the precision. Only a few bits can not precisely express the floating point character of the synaptic weights value.

For the evolution can be used any of the evolutionary algorithm. But when genetic algorithms are chosen, evolution does not have to necessarily lead to desired results.

It is because the crossover recombination operator can easily destroy detected features in ANN [3]. Another problem with binary representation and genetic algorithms is connected to the known permutation problem as described in section 4.1.1. Therefore it is not recommended to use the GA with the crossover recombination operator to evolve the configuration.

4.2.2 Real Number Representation

When using real numbers to represent synaptic weights we do not have to make any decisions how to represent genes. Chromosome is then a vector of real values.

But because the real numbers are used, standard crossover recombination operator can not be used. Thus other, special operators, can be designed but using evolutionary programming or evolutionary strategies seems to be better way [21]. Authors of mentioned work also proposed following evolutionary programming algorithm to evolve synaptic weights.

The chromosome is a pair of real valued vectors $(\mathbf{w}, \eta)$ of size n. $\mathbf{w}$ is the configuration of the network (vector of all synaptic weights) and η is a variance vector for Gaussian mutation. The offspring is created using following equations:

$$\mathbf{w}'(j) = \mathbf{w}(j) + \eta'(j)N_j(0,1) \tag{4.2}$$

$$\eta'(j) = \eta(j)exp(\tau' N(0, 1 + \tau N_j(0,1))) \tag{4.3}$$

Equation 4.2 refers to newly created configuration and 4.3 refers to newly created Gaussian mutation's variance vector.

Item j denotes the range of $1 \to n$. Items $w(j), w(j)', \eta(j)$ and $\eta'(j)$ denote the j_{th} component of vectors $\mathbf{w}, \mathbf{w}', \eta$ and η'. $N(0,1)$ denotes a normally distributed one-dimensional random number with mean zero and variance one. $N_j(0,1)$ indicates that the random number is generated anew for each value of j. The parameters τ and τ' are commonly set to $(\sqrt{2\sqrt{n}})^{-1}$ and $\sqrt{2n}^{-1}$. $N_j(0,1)$ in equation 4.3 may be replaced by Cauchy mutation for faster evolution.

To create the new generation was proposed following selection scheme: For each chromosome from the original population $(\mathbf{w}, \eta)$ and offspring $(\mathbf{w}', \eta')$ choose q opponents uniformly at random from the original population and offspring. Chromosome with higher fitness wins. The new generation is then composed of chromosomes with the highest number of wins.

4.3 Simultaneous Evolution of the Topology and Configuration

In previous sections we have mentioned two approaches to evolve artificial neural networks. First uses evolution to discover the topology, second to replace the learning process.

What we have not mentioned is problem often referred as one-to-many mapping between genotype and phenotype. Lets consider two similar architectures and learning algorithms. Using standard training process can lead to different behaviour of the final network. That is caused by the learning process. Every learning algorithm sets synaptic weights randomly as the first step of the process. Adapting synaptic weights then leads to different final results what leads to inaccuracies in the evolution process.

To avoid this problem we would have to amend the learning process to use the same initial configuration and send the learning data to the input in the same order for all networks within the generation. Another option is to simultaneously evolve the topology and configuration (synaptic weights).

Evolving the the topology and configuration simultaneously avoids the mention problem because each chromosome will have always the same fitness. When only topology was evolved, one chromosome could have different fitness. That is because the reconstructed network has to be trained. After the learning process finishes the network can have different error-neous. That's why the evolution process results into inaccuracies.

In section 4.1.1 we have used the direct encoding to evolve the topology. To evolve also synaptic weights we have to change the gene representation. That results into the change of the chromosome representation. Instead of values (0, 1) the gene has the value equal to the value of synaptic weight. We can use the binary encoding or real value representation. In order to create the chromosome we use the same procedure as in section 4.1.1.

Evolution process is then similar to the one described in section 4.1.1. Also using genetic algorithms with crossover reproduction operator again leads to permutation problem or loss of the feature (in case of use of crossover recombination operator).

4.4 Evolution of Learning Rules

Evolution of learning rules is referred to an area of learning algorithms evolution. As we have stated earlier in chapter 2 there are multiple learning algorithms for the ANN. Different learning algorithms have different set of attributes to set before the learning process can start. Some of them are more sensitive than others but all of them have influence on the final result.

Evolution of learning rules can be divided into two groups. First is the **evolution of the algorithms attributes** only, second relates to the **usage of learning rules as the whole**.

Similarly to the parametric representation of the evolution of ANNs architecture or to the evolution of transfer functions, only attributes of learning algorithms are evolved. In that case gene represents a single attribute, e.g. momentum of BP learning algorithm. In order to represent the gene can be chosen suitable static representation and also binary representation can be used. In that case we can use any kind of EAs leading to no previously mentioned problems.

Until now we relied only on one type of learning algorithm. But for different problems they can have different results. So another method has been introduced. This method allows multiple learning algorithms to be used within the population and they are evolved simultaneously.

Lets consider the MLP network. We have mentioned backpropagation learning algorithm. But there are also its deviations, such as momentum backpropagation, etc. The chromosome would consists of one gene. The gene would be a code of one algorithm. For example standard backpropagation would have a code 0 and momentum backpropagation 1.

But such an evolution is very limited. Therefore it is suggested to combine it with other possibilities of the EANN.

The problem here is the gene representation. Different algorithms have different attributes. Proper encoders have to be used what can prolong the evolution process. Also only special mutation recombination operator can be used. In most cases are used evolutionary strategies.

4.5 Feature Selection

Some features in the learning data set can be redundant or noisy which can lead to longer and / or inaccurate training process. The feature selection process tries to find the near optimal learning data set. It can be applied to the artificial neural networks and also to any other kind of machine learning techniques.

The gene's representation is mostly binary where 1 indicates the presence and 0 the absence in the final learning data set. One gene represents one feature and the chromosome is concatenated binary string of all features in the learning data set.

Lets consider data set with following features [2]: class, age, meno-

2. example taken from Breast Cancer Data Set;

pause, tumor-size, inv-nodes, node-caps, deg-malig, breast, breast-quad, irradiat, weather, month. Corresponding chromosome with all features set to be present in learning process is shown on figure **??**. In this data set are two features which with high probability have no addition in order to find out if the person has or does not have the breast cancer. Corresponding chromosome with these two features excluded from the learning process is shown on figure 4.13.

11111111111	11111111100
(a) Chromosome with all features present in the learning process	(b) Chromosome with two features excluded from the learning process

Figure 4.13: Feature selection chromosomes example

To evolve the features can be used any kind of evolutionary algorithm. Also using crossover recombination operator will not break any hidden feature in ANN and does not lead to permutation problem.

But the problem is that this method suffers to one-to-many genotype to phenotype mapping. As we have mentioned in previous section, this can lead to inaccuracies in evolution process. Evolving only features is therefore not recommended and should be used together with the evolution of configuration.

4.6 Use of the Generation

Presented methods use the information obtained in the fittest chromosome in the last iteration of the population. But whole generation of chromosomes can provide more information than a single chromosome.

There are several ways how to combine the information obtained in the generation into single combined system. First and the simplest method is a **majority voting**. In the majority voting every chromosome in population has the same strength of the vote. Final ensemble output is the output of the most voted, in our case, artificial neural network. If there is a tie, the most precise ANN is chosen to be the output.

http://archive.ics.uci.edu/ml/datasets/Breast+Cancer

Lets consider a population of four artificial neural networks and som problem to solve. Two of the ANNs have the same output value and remaining two have different. As the final output value is then the output value of those two ANNs.

Main disadvantage of this method is that all chromosomes are treated equally. If there is a chromosome with high error rate it can worse the output ensemble. On the other hand this method is very simple and easy to implement. No extra computation time is needed to get the final output.

Rank-based linear combination takes into the consideration the differences between chromosomes. Also it does not require extra computational cost as it uses the fitness information to compute a weight of the vote. Authors of [22] proposed following equations to compute the weight:

$$w_i = \frac{exp(\beta(N+1-i))}{\sum_{j=1}^{N}(exp(\beta j))} \tag{4.4}$$

w_i is the i_{th} weight of the i_{th} chromosome, N is the population size and β is a scaling factor. It should be set in interval from 0 to 1. The output value of the ensemble is calculated as follows:

$$O = \sum_{j=1}^{N} w_j o_j \tag{4.5}$$

O is the final output value and o_j is the output value of the j_{th} chromosome.

Lets consider the same example as above. Consider also the fitness of the chromosomes. Those two networks with the same output value have also small fitness. Applying the equations 4.4 and 4.5 can weaken the strength of their vote and the output value can different from theirs.

Previous methods used whole population to get the output. But also **subset of the population** can be used as an ensemble. The space of possible subsets is huge ($2^n - 1$), therefore for large populations can be searching the optimal subset very exhaustive. To search the near optimal subset can be used the evolution. This method can lead to more precise and better generalization EANN ability as the worst individuals in the population are excluded from the ensemble.

5 Neuroph

Neuroph is a Java open source framework designed to develop Artificial Neural Networks released under Apache 2.0 license [1]. It contains an implementation for most of the mainstream ANNs and learning algorithms, such as Multilayer Perceptron network and Backpropagation learning algorithm [18]. We use the framework's latest version currently available, which is version 2.8. The technical documentation can be found on the internet [2].

We use Neuroph for the ANN part of the framework. In following sections we describe its basic components.

5.1 MultiLayerPerceptron

The multilayer perceptron artificial neural network is represented by the class `MultiLayerPerceptron`. It is inherited from the general neural network class called `NeuralNetwork`.

In order to create an instance of the MLP object we have to provide attributes such as the number of neurons in layers and a set of settings for neuron initialization, such as the type of the input [3] and transfer functions. The number of neurons in layers is represented by the structure `List<int> neuronsInLayers` [4]. The set of settings is represented by the class `NeuronProperties`.

`MultiLayerPerceptron` contains all required structures of the network, such as the topology, which is organized into layers and the configuration of the network.

Layers are represented by the class `Layer` and it is an array of `Neuron` objects. Between two neurons it is created a connection (`Conne ction`) and each connection has a weight (`Weight`). `Neuron` is a representation of the formal neuron (see **??**).

1. http://www.apache.org/licenses/LICENSE-2.0.html
2. http://neuroph.sourceforge.net/documentation.html
3. Input function refers to the *inner potential* as described in section 2.1.
4. array of integers where each number represents the number of neurons in one layer

5.2 BackPropagation

Backpropagation learning algorithm is represented by the class `BackPropagation`. It is inherited from the class `LearningRule`. We can find the complete list and hierarchy of Neuroph's learning algorithms on the internet [5].

`BackPropagation` provides an option to set the algorithm's properties, such as the learning rate, maximum error or maximum number of learning iterations after which the learning process stops.

Learning algorithm classes are designed to implement the learning process. The process stops when it satisfies one of the stop conditions. Stop conditions are defined by `StopCondition` interface. In the current version of the framework are implemented three stop conditions: `MaxErrorStop`, `MaxIterationsStop` and `SmallErrorChangeStop`.

5.3 DataSet

Instances of the class `DataSet` are used to represent the training, validation or testing set. The input data of every Neuroph's network is an instance of the `DataSet` class.

In order to create the training and validation data sets, we have to provide two attributes to the constructor of `DataSet`, sizes of the input and output vectors. In order to send the input data, we provide only one attribute, size of the input vector.

There are multiple ways how to load the data into `DataSet`. In our implementation we use the load via a plain text file. The file has to be in CSV format. The delimiter for the CSV file can differ from the CSV specification [6]. It is provided to the `DataSet` when it start to read the data from the file (method `createFromFile`).

5. http://neuroph.sourceforge.net/images/uml/LearningRulesClassDiagram.jpg
6. http://tools.ietf.org/html/rfc4180

6 EANN Framework

As part of our thesis we have designed and implemented a standalone framework application. The framework is split into two parts:

1. jEvolution - the implementation of the evolutionary algorithms;
2. jNeuroph - the bridge between jEvolution and Neuroph.

6.1 jEvolution

In respect to the chapter 3 we split this section in three sub sections: components, evolution process and evolutionary algorithms.

6.1.1 Components

From the section 3.1 we know that in order to create an EA we have to define following components: population, chromosomes, genes and the fitness function. Genotype and phenotype serve only as additional info and do not have to be implemented. Our framework implements these components as follows.

The class **`Population`** contains a list of chromosomes. This class provides a number of methods to manipulate the chromosomes, such as adding to and removing from the population and replacement of the whole population.

The class **`Chromosome`** contains a list of genes and stores the fitness information. It contains a number of methods to manipulate the genes, such as adding to and removing from the chromosome and specific gene replacement. It is also able to initialize the chromosome to initial, random, value. That means to set all its chromosome to random values. The `Chromosome` can be also cloned using method `clone`. It performs the deep copy [1] of the chromosome.

In chapters 3 and 4 we have stated that the gene representation can vary from case to case. That is why the framework contains multiple representations of genes (see figure A.5). All genes implements the interface **`IGene`**. It provides the ability to perform the deep copy

1. http://www.cs.utexas.edu/ scottm/cs307/handouts/deepCopying.htm

of the gene (method `clone`) and it also provides the mutation ability. The mutation is different for each representation of the gene. For example the binary gene changes its value from 1 to 0 and vice versa. The real valued gene changes its value within the provided range (often referred as uniform mutation).

In those chapters we have also stated that the the fitness function determines how is the problem (application) evaluated and how it is defined. That is why the implementation of fitness function is the matter of concrete applications which use jEvolution framework. But the structure of the fitness function has to implement the interface `IFitnessFunction`. jEvolution uses this interface in the evolution process.

6.1.2 Evolution Process

The evolution process is controlled by the class `EvolutionChamber`. It stores the configuration of the evolution process, such as the maximal number of iterations (epochs), number of current iteration and the threshold value to determine if the possible solution is near optimal. `EvolutionChamber` contains two structures: an `EvolutionaryAlgorithm` and the `Population`. We describe the `EvolutionaryAlgorithm` in the next section.

In order to start the evolution process we have to provide the `EvolutionaryAlgorithm` and the threshold value. The threshold value determines the value of the fitness which is considered to be near optimal to the solution (see 3.2).

The evolution process starts by calling the method `startEvolution`. After that it is created an initial population which is evolved until stopping criteria (see section 3.2) are not hit.

By default the `EvolutionaryAlgorithm` does not have set any restriction to the number of iteration. We can set it calling the method `limitEpochs` and unset it calling `unlimitEpochs`.

When the evolution process finishes we can get the current population by calling the method `getCurrentPopulation`. To get the number of iterations serves the method `getEpoch`.

6.1.3 EvolutionaryAlgorithm

`EvolutionaryAlgorithm` is the main abstract class which implements the common functionality for each type of the evolutionary algorithm, see picture A.2

The relation between `EvolutionChamber` and `EvolutionaryAlgorithm` is that while `EvolutionChamber` drives the evolution process, `EvolutionaryAlgorithm` implements it.

Individual evolutionary classes implements only the methods which differ from the common approach. As an example we can take the recombination process. In order to create the offspring the GA uses the crossover operator followed by the mutation. The EP and ES use only the mutation.

`EvolutionaryAlgorithm` contains following structures: `IApplication`, `FitnessEvaluator`, `IFitnessFunction`, `IReproductionModel` and `ISelector`.

`IApplication` is the interface which represents our problem to solve. It provides three basic methods which have to be implemented:

- `getSampleChromosome` - Creates a chromosome initialized to the random value.
- `toChromosome` - Converts given application structure into the chromosome information.
- `toInstance` - Converts the chromosome information into the application structure.

We have described the **`IFitnessFunction`** in section 6.1.1. The **`FitnessEvaluator`** evaluates each chromosome by its fitness using the `IFitnessFunction`. The `IFitnessFunction` also have to provide the information about the solution's target cost. The cost can be minimal or maximal (method `getCost`).

Lets say we want to find the topology of the network. In section 4.1.1 we have outlined that we can look for the minimal or maximal topology. This is reflected by the solution's target cost.

Reproduction ability is covered by the **`IReproductionModel`** interface. It is implemented as per figure A.3.

The **`ISelector`** interface takes care about the selection process. It is designed and implemented as per figure A.4.

6.2 jNeuroph

jNeuroph implements the provided interfaces from the jEvolution framework from the previous section (such as `IApplication` and `IFitnessFunction`). It also implements a set of classes which help to transform the Neuroph's `NeuralNetwork` into the `Chromosome`. We can see the jNeuroph's basic class diagram on figure A.6.

6.2.1 IBuilder, IExtractor, ITransformer

Classes which implement the **`IExtractor`** interface serve to extract the information from the network. The interface provides one public method which has to be implemented, `extract`. The return type of this method depends on the feature we want to extract. It can be an array of doubles (in case of extracting the configuration in form of synaptic weights), matrix of booleans (representation of the architecture's topology) or some map (map represents the set of attributes of the learning algorithm or transfer function). Figure A.7 shows us the current implementation of all extractors.

The extracted information can be used in two ways. From the information can be built a chromosome or an artificial neural network. The chromosome is evolved in the evolution process. The ANN is the resulting application from the evolution process.

To build the chromosome or ANN serves an **`IBuilder`** interface. The `IBuilder` interface provides one public method to implement, `build`. It turns the extracted information into the required structure.

Figure A.8 shows all implemented builders which serve to purposes of our thesis.

`ITransformer` interface transforms the chromosome back into the information. It provides one public method to implement, `trans form`. We can see the transformer's basic class diagram on figure A.9.

Each application inherited from the class `NeurophApplication` follows the following procedure:

1. extract the information from the artificial neural network.

2. from the extracted information build a chromosome
3. perform the evolution
4. transform the chromosome into information
5. from the transformed information build an artificial neural network

This process has to maintain the integrity of the application in respect to the information being evolved. That means if no alteration will happen during the evolution, the network before the start and after the end of this process should be the same. Of course, when we extract only partial information, some amount of loss of information is expected.

For example we extract the full architecture with synaptic weights. The network in the end of the process will have the same topology and the same synaptic weights as it had on the beginning of the process. But if we extract only the topology of the network, in the end of the process the configuration of the network will be different.

Limitations

In section 4.1.1 we have stated that we can reduce the size of the chromosome for the feedforward networks because of the nature of their topology. In section 2.2.1 in paragraphs about the topology we have stated that the connections in MLP network are strictly made between neighbouring layers, $l_{i-i} \rightarrow l_i; 1 < i \leq n$.

The size of the input and output layers is predetermined by the size of the input and output data from the data set. If the EA would alter the topology in order to create the connection between neurons in the same layer, the topological principle would be broken. In respect to this fact we restrict the size of the chromosome to exclude this information. The other option would be ignoring it (hardcoded the input and output layer size) but that would result into the evolution process inefficient.

6.2.2 NeurophApplication

The `NeurophApplication` class implements the `IApplication` interface (see section 6.1). `NeurophApplication` is designed to provide the basic functionality for every type of the EANN described in chapter 4.

To demonstrate the EANN functionality we have implemented following applications: `ConfigurationEngineer`, `TopologyArchitect` and `TopologyEngineer`.

`ConfigurationEngineer` is an application designed to evolve the network's configuration. It stores the basic information about to network in order to reconstruct it, such as the number of layers and number of the neurons in layers. It uses following implementations of described interfaces in previous section:

- `ConfigurationExtractor` - extracts the array of all synaptic weights.
- `ConfigurationChromosomeBuilder` - from the extracted array builds a chromosome. One gene of the chromosome represents one value from the array.
- `DoubleArrayTransformer` - transforms the chromosome into the array of doubles.
- `ConfigurationNetworkBuilder` - creates the network according to the information stored in the application and sets its synaptic weights according to the transformed array.

`TopologyArchitect` is an application designed to design the network's topology. It uses following implementations of described interfaces in previous section:

- `TopologyExtractor` - extracts the two dimensional array of all connections in the network as described in section 4.1.1.
- `TopologyChromosomeBuilder` - from the extracted array builds a chromosome. This implementation is specific for the feedforward networks as described in section 4.1.1 so the size of the

chromosome is reduced. The builder has also another limitation and that is the size of the input and output vectors. Feed-forward networks don't allow the connections between neurons in the same layer. Therefore neurons in the input and output layers are excluded from the chromosome.

- `BooleanUpperMatrixTransformer` - transforms the chromosome into the two dimensional array of booleans.
- `ConfigurationNetworkBuilder` - creates the network according to the evolved topology, initializes the synaptic weights to random values and triggers the normal learning process of the artificial neural network.

`TopologyEngineer` is an application designed to evolve to network's configuration and topology simultaneously. It uses following implementations of described interfaces in previous section:

- `ConfiguredTopologyExtractor` - extracts the two dimensional array of all connections with their synaptic weights.
- `ConfiguredTopologyChromosomeBuilder` - from the extracted array builds a chromosome. The implementation suffers to similar restrictions as the implementation of `Topology ChromosomeBuilder`.
- `DoubleArrayTransformer` - transforms the chromosome into the two dimensional array of doubles.
- `ConfigurationNetworkBuilder` - creates the network according to the evolved topology and sets its synaptic weights according to the transformed array.

7 Experiments

As part of this thesis we present the experiments that corresponds to the implementation described in chapter 6. This chapter is organized as follows. At first we describe the experiments procedure. Then we describe results of experiments with data set obtained from the UCI machine learning repository [1]. Then we perform the same experiments with the data we have obtained from the company ReeGo development s.r.o. The structure of the data is described in corresponding sections.

7.1 Procedure

We have trained artificial neural networks with the following methods:

- standardlearning process;
- evolving the configuration;
- designing the topology;
- designing the topology simultaneously with evolving the configuration.

The objective of our experiments is to find out if use of the evolution is beneficial. To evolve the network we have used evolutionary programming with uniform mutation.

In our experiments we measure following attributes:

- number of neurons in layers;
- number of connections;
- learning rate;
- total error (training set);

1. http://archive.ics.uci.edu/ml/datasets.html

- total error (testing set);
- regression error (training set);
- regression error (testing set);
- number of iterations;
- time of the learning / evolution process in miliseconds.

Not every attribute is valid for all methods. When the evolution of the configuration is involved in the process it does not matter whether the data set is training or testing. All synaptic weights are evolved therefore all data introduced to the network are basically testing. Also standard learning process assumes that all possible connections in the network exist. The evolution of configuration also works with all connections in the network.

The experiments were performed on a machine with following configuration:

- CPU - Intel Pentium i3 370M 2.66 GHz;
- memory - 4GB DDR3 1333 MHz SDRAM;
- storage - 500GB 7200rpm;
- OS -: Linux 3.2.0-4-amd64; Debian 3.2.51-1 x86_64 GNU/Linux
- JDK version - 1.7.0_45.

The experiments were performed ten times from which was derived the median value. In case of topology design were chosen the solutions with minimal error.

7.2 Computer Hardware Data Set

The data was obtained from Tel Aviv University. The authors of the data set are Phillip Ein-Dor and Jacob Feldmesser. The data set contains 209 instances with the total of 9 attributes and it does not contain any missing values. The data were donated on 1987-10-01.

The aim of this experiment is to predict the relative performance based on the following attributes:

1. vendor name (string)
2. model name (string)
3. machine cycle time in nanoseconds (integer)
4. minimum main memory in kilobytes (integer)
5. maximum main memory in kilobytes (integer)
6. cache memory in kilobytes (integer)
7. minimum channels in units (integer)
8. maximum channels in units (integer)
9. published relative performance (integer)
10. relative performance from the original article (integer)

The first two attributes are irrelevant to our experiments and therefore are excluded from the data set. The rest of the attributes must be normalized as they contain high range of values. That can lead to ineffective learning process and poor generalization ability. The last attribute in the data set is the desired value which will be compared with the computed output from the network.

In order to normalize the data we use following formula:

$$A = \frac{B - min(B)}{max(B) - min(B)} \times (C - D) + D \tag{7.1}$$

Where A is the standardized value, B is the original value and C and D determines the range in which we want our value to be (C = 1 and D = 0).

The detailed description of this data set can be found on the internet [2].

In our first experiment we have trained the network by standard learning algorithm (backpropagation). We have set the maximal iterations to 100 and maximum error rate to 0.01. Our goal have been to find the network with the least error. We have tried 16 runs. In the table 7.1 are presented two best and two worst results.

2. http://archive.ics.uci.edu/ml/datasets/Computer+Hardware

	1	2	15	16
Neurons in layer	[6, 3, 1]	[6, 1, 1]	[6, 7, 1]	[6, 6, 1]
Learning rate	0.2	0.1	0.2	0.1
Total Error (training)	0.0088	0.0090	0.0098	0.0098
Total Error (testing)	0.0073	0.0074	0.0086	0.0086
Regression Error (training)	0.0194	0.0468	0.0090	0.0114
Regression Error (testing)	0.0044	0.0350	0.0071	0.0048
Iterations	4	12	3	4
Time	58	67	62	65

Figure 7.1: Backpropagation learning - results

	1	2	3	4
Neurons in layer	[6, 3, 1]	[6, 1, 1]	[6, 7, 1]	[6, 6, 1]
Total Error	0.0088	0.0087	0.0096	0.0093
Regression Error	0.0194	0.0438	0.0157	0.0616
Iterations	16.5	5.33	32.14	4.23
Time	925.21	501.52	985.23	750

Figure 7.2: Evolution of the configuration - results

From the experiments seem to be the best network with three neurons in the hidden layer and the learning rate 0.2. The worst network had seven neurons in hidden layer and the learning rate was 0.1.

For the networks from the first experiment we tried to find the satisfying configuration using evolution. Our objective have been to find out if the network's error can be improved. We have set the population size to 100 and limited the evolution process to 100 epochs. We have repeated our experiments 10 times with one configuration. The table 7.2 shows the average values of individual runs.

By the configuration's evolution we have resulted into only a bit better results. Also the execution time was longer.

In our third experiment we have tried to discover the topology of the network automatically. To train the network we have used the learning algorithm's parameters of the network which performed best in our first experiment. The population size and number of epochs

	1	2	19	20
Neurons in layer	[6, 7, 6, 1]	[6, 6, 7, 1]	[6, 4, 1]	[6, 3, 1]
Number of connections	40	41	11	10
Total Error (training)	0.0076	0.0079	0.0094	0.0095
Total Error (testing)	0.0071	0.0071	0.0086	0.0087
Regression Error (training)	0.0493	0.0533	0.0430	0.0467
Regression Error (testing)	0.0416	0.0458	0.0309	0.0383
Iterations	1	1	1	1
Time	1422	1236	130355	150369

Figure 7.3: Topology design - results

were set to the same parameters as in our previous experiment. We have set the maximum number of neurons to 20. The minimum number of neurons have been 10. The objective of this experiment have been to find out if we can find a topology with better results than we have found so far. We have executed the method for 10 times per number of neurons and the best two and worst two results are in the table 7.3.

As population was big enough we were able to find the result in one epoch. That was given by the learning process followed after the epoch ended. The stopping criteria to end the learning process matches the stopping criteria of the evolution process (in form of the error-neous). On the other hand, because of the learning process was the evolution time significantly longer than in previous methods.

Most of the times we have discovered the topologies with 2 hidden layers with 7 and 6 or 6 and 7 neurons in layers. The worst networks have had 1 hidden layer and small connectivity. Also it has taken long time to find the near optimal solution. But our best networks have had lower total error than our previous findings.

Our final experiment was involved the simultaneous evolution of the topology and configuration. The population size and epochs restriction were left unchanged to the previous experiments. The results are in the table 7.4.

It have taken more epochs and more time to discover the satisfying network. Results are similar to discovering only topology. But the difference between best and worst result is the biggest we have

	1	2	19	20
Neurons in layer	[6, 6, 7, 1]	[6, 6, 2, 1]	[6, 6, 1]	[6, 5, 1]
Number of connections	37	37	18	22
Total Error	0.0077	0.0079	0.0181	0.0264
Regression Error	0.0457	0.0533	0.0272	0.0976
Iterations	12	8	100	100
Time	1715	1388	2790	3288

Figure 7.4: Simultaneous evolution of the topology and configuration - results

encountered so far. The worst results have always hit the maximal epochs stopping criteria.

7.3 Flat Prices

The data was obtained from the company ReeGo Development s.r.o. The data set is an extract from the real estate CRM system ReeGo. It contains nine attributes and 20,475 instances with no missing values. The data set was donated on 2013-12-01.

The aim of this experiment is to predict the price of the flat based on the following attributes chosen by the real estate expert:

1. flat kind (integer)
2. building type (integer)
3. building condition (integer)
4. floor number (integer)
5. floors (integer)
6. floor area (float)
7. balcony or loggia area (integer)
8. ownership (integer)
9. heating (integer)

	1	2	19	20
Neurons in layer	[10, 10, 4, 1]	[10, 9, 1]	[10, 2, 1]	[10, 1, 1]
Number of connections	61	45	35	12
Total Error	0.0044	0.0049	0.0083	0.0967
Regression Error	0.0022	0.0081	0.0275	0.4313
Iterations	15	5	100	100
Time	6388	4293	20790	31685

Figure 7.5: Simultaneous evolution of the topology and configuration - results

10. advert price (float)

We can find the description of the attributes and their complete list on the internet [3].

Similarly to the previous experiments the values in the data set are normalized using the equation 7.1.

We have started our experiments with the simultaneous evolution of the topology and configuration and the topology design. We have used the population of size 100, maximum number of epochs set to 100 and the number of neurons from 12 to 15. Our objective have been to discover interestting topologies which we could use in next experiments. The results from experiments are in tables 7.5 and 7.6.

From the results we can see that the most successfull topologies have been [10, 10, 4, 1] and [10, 9, 1]. Those two we have used in following experiments with the configuration evolution and standard learning process.

We have run the process 10 times per architecture. The results in tables 7.7 and 7.8 showes us the average values.

3. http://www.reego.cz/pro-techniky/reego-import/xml-schema/detail-nabidky/

	1	2	19	20
Neurons in layer	[10, 10, 4, 1]	[10, 9, 1]	[10, 4, 1]]	[10, 1, 1]
Number of connections	50	24	19	7
Total Error (training)	0.0037	0.0044	0.0055	0.0056
Total Error (testing)	0.0047	0.0055	0.0064	0.0064
Regression Error (training)	0.0009	0.0010	0.0050	0.0005
Regression Error (testing)	0.0040	0.0060	0.0003	0.0003
Iterations	1	1	1	1
Time	26721	39684	119544	109997

Figure 7.6: Topology design - results

Neurons in layer	[10, 10, 4, 1]	[10, 9, 1]
Total Error	0.0092	0.0040
Regression Error	0.1036	0.0038
Iterations	5.2	6.4
Time	6256.21	8462.52

Figure 7.7: Evolution of the configuration - results

Neurons in layer	[10, 10, 4, 1]	[10, 9, 1]
Learning rate	0.2	0.2
Total Error (training)	0.0068	0.0048
Total Error (testing)	0.0075	0.0056
Regression Error (training)	0.0740	0.0454
Regression Error (testing)	0.0693	0.0404
Iterations	58.2	64.9
Time	510.6	401.5

Figure 7.8: Backpropagation learning - results

8 Discussion and Future Work

8.1 Discussion

In our work we have outlined couple of possible complications in order to build an artificial neural networks. We have also outlined their common use with evolutionary algorithms.

In order to create an artificial neural networks with satisfying results we have to have, in most cases, expert knowledge. Designing the proper topology can be hard task to achieve. Also learning algorithms require a lot of attributes to set. Setting them properly without any experience on this field it can be the matter of coincidence. As we have discussed in our thesis, the evolution can help to solve this problems.

Almost each data set have its specifics. In order to investigate the different data sets we have to create networks with different parameters. But when the network is evolved, one evolutionary implementation can be used with different data sets. The task of the EA remains the same, find the near optimal solution.

Some learning algorithms, such as the backpropagation and other gradient based algorithms, can stuck in local optima. The evolution can help to prevent this problem as it can be used to find the satisfying configuration.

8.2 Future Work

This thesis serves as first draft of bigger and more complex application framework for the artificial neural networks. We have implemented three techniques: the topology design, evolution of the configuration and the simultaneous evolution of the topology and configuration. In our experiments we have used evolution programming with uniform mutation.

This is only a small part from the possibilities outlined in this thesis. We have not covered experiments with different types of evolutionary algorithms such as genetic algorithms or evolution strategies. Also different types of the mutation, such as a Gaussian or Cauchy mutation, can lead to different results.

To determine the fitness of the network we have used only one fitness function. Using different fitness functions can lead to different results. This possibility can be investigated in the future, especially in case of the topology design. Learning process followed by the end of evolution epoch resulted into only one epoch of the evolution. Having different function measuring different attribute of the network, such as the complexity of the topology, can bring better results in discovering the topology.

Also only one area of the problems solvable by artificial neural networks were inspected. Except prediction (regression) it can be the classification or clustering problem.

In our thesis we focus on the feedforward networks, more specifically multilayer perceptron. But there are more types of ANNs which can be used with the evolution such as a Hopfield network, RBF networks, LVQ networks, etc.

9 Conclusion

In this work we have collected the information about evolutionary artificial networks. We have chosen three techniques, such as evolution of the configuration, the topology design and the simultaneous evolution of the topology and configuration, and we have implemented them. We have used our implementation to train the network with three different data sets.

Our objective was to find the network with the minimal error and conclude if evolution is beneficial. We have demonstrated that the evolution can be used together with the artificial neural networks. Although from the results of our experiments is not clear if the evolution was beneficial.

But by our experiments we have covered only part of the possibilities of the EANNs. Further research and experiments are required.

A Class Diagrams

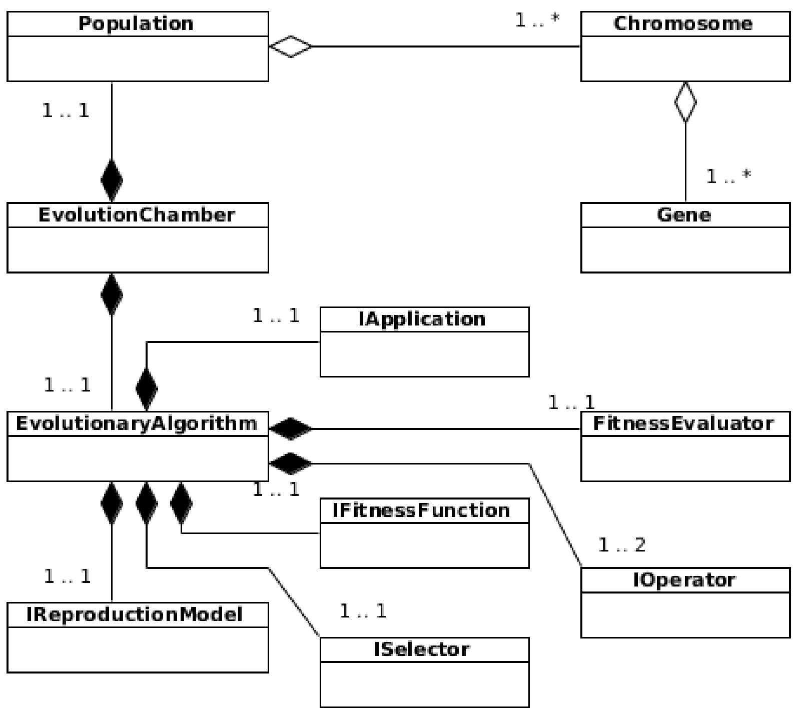


Figure A.1: jEvolution

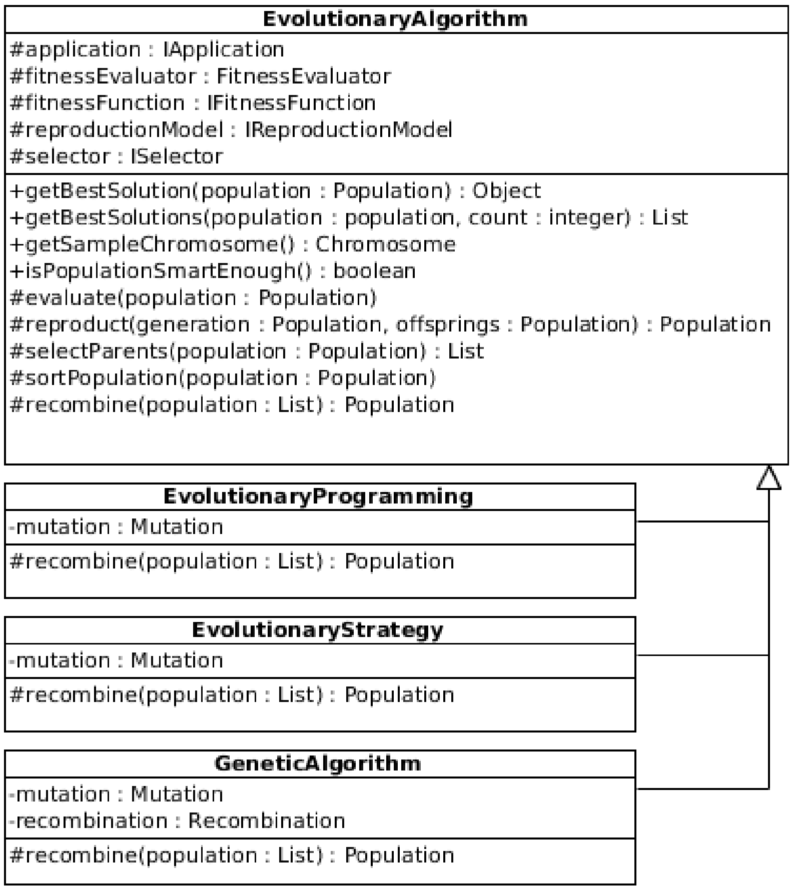

EvolutionaryAlgorithm
#application : IApplication
#fitnessEvaluator : FitnessEvaluator
#fitnessFunction : IFitnessFunction
#reproductionModel : IReproductionModel
#selector : ISelector
+getBestSolution(population : Population) : Object
+getBestSolutions(population : population, count : integer) : List
+getSampleChromosome() : Chromosome
+isPopulationSmartEnough() : boolean
#evaluate(population : Population)
#reproduct(generation : Population, offsprings : Population) : Population
#selectParents(population : Population) : List
#sortPopulation(population : Population)
#recombine(population : List) : Population
EvolutionaryProgramming
-mutation : Mutation
#recombine(population : List) : Population
EvolutionaryStrategy
-mutation : Mutation
#recombine(population : List) : Population
GeneticAlgorithm
-mutation : Mutation
-recombination : Recombination
#recombine(population : List) : Population

Figure A.2: Evolutionary Algorithms

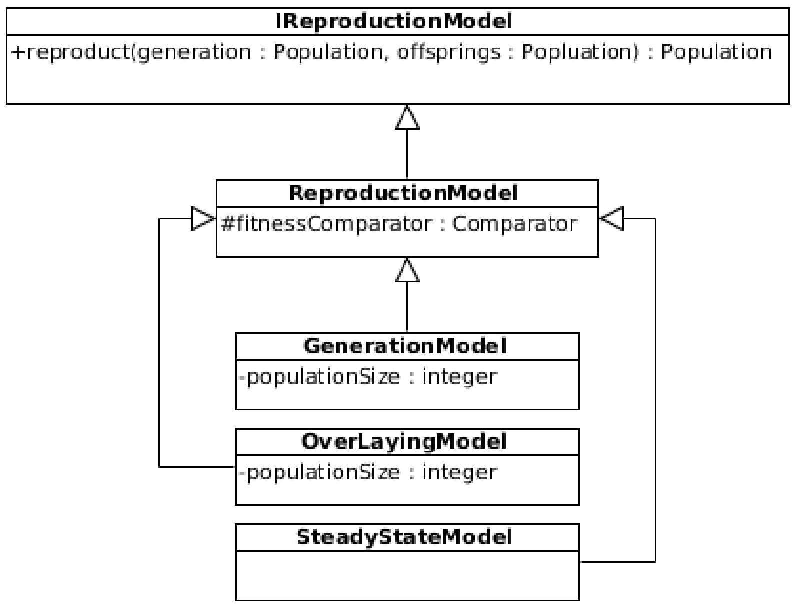

IReproductionModel
+reproduct(generation : Population, offsprings : Popluation) : Population
ReproductionModel
#fitnessComparator : Comparator
GenerationModel
-populationSize : integer
OverLayingModel
-populationSize : integer
SteadyStateModel

Figure A.3: Reproduction Models

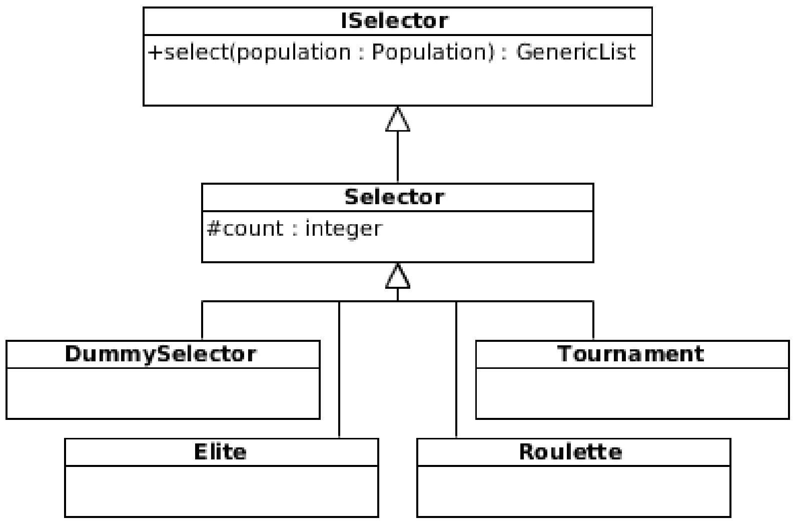


Figure A.4: Selectors

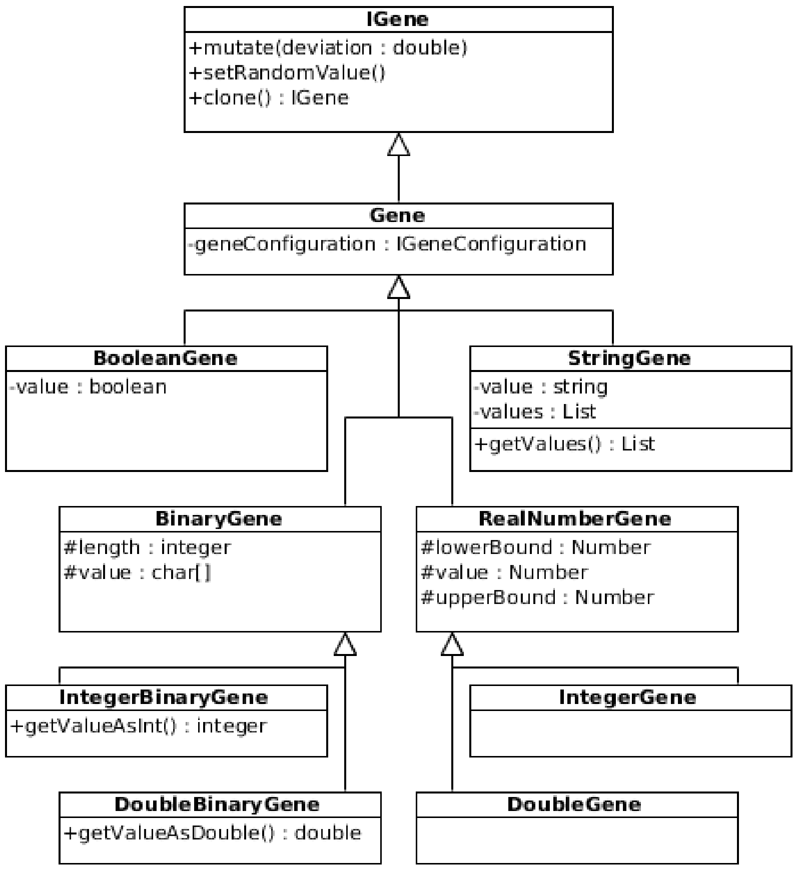


Figure A.5: Genes

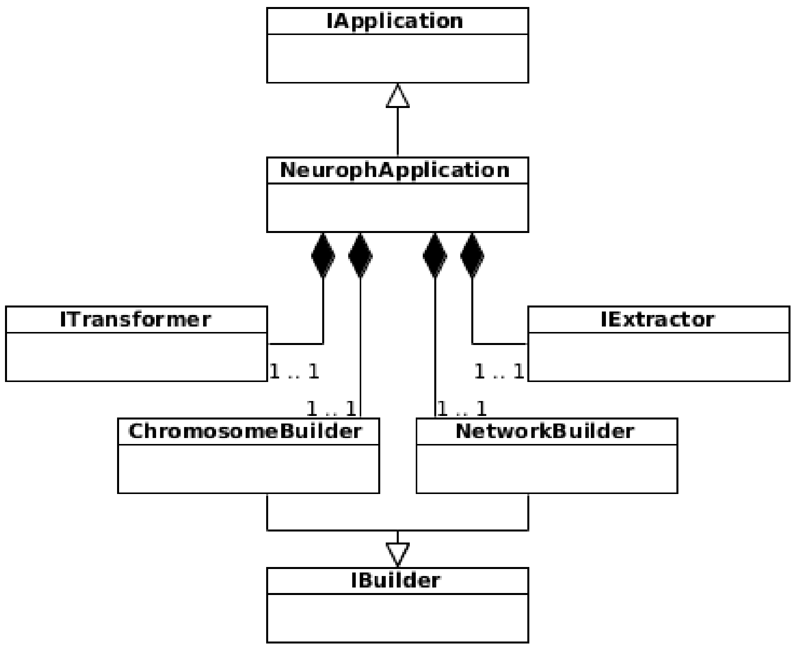


Figure A.6: jNeuroph

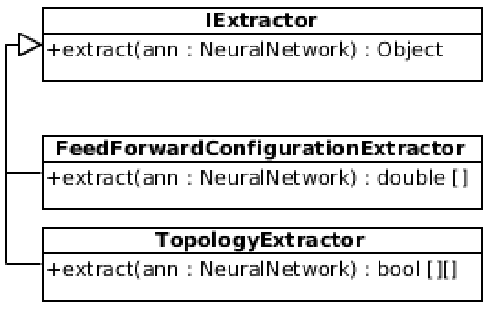

IExtractor
+extract(ann : NeuralNetwork) : Object
FeedForwardConfigurationExtractor
+extract(ann : NeuralNetwork) : double []
TopologyExtractor
+extract(ann : NeuralNetwork) : bool [][]

Figure A.7: Extractors

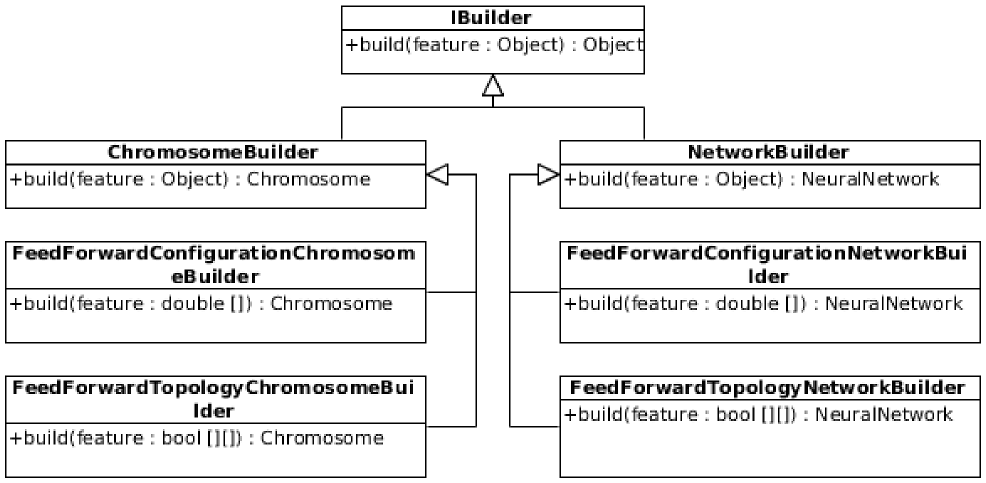

IBuilder
+build(feature : Object) : Object
ChromosomeBuilder
+build(feature : Object) : Chromosome
NetworkBuilder
+build(feature : Object) : NeuralNetwork
FeedForwardConfigurationChromosomeBuilder
+build(feature : double []) : Chromosome
FeedForwardConfigurationNetworkBuilder
+build(feature : double []) : NeuralNetwork
FeedForwardTopologyChromosomeBuilder
+build(feature : bool [][]) : Chromosome
FeedForwardTopologyNetworkBuilder
+build(feature : bool [][]) : NeuralNetwork

Figure A.8: Builders

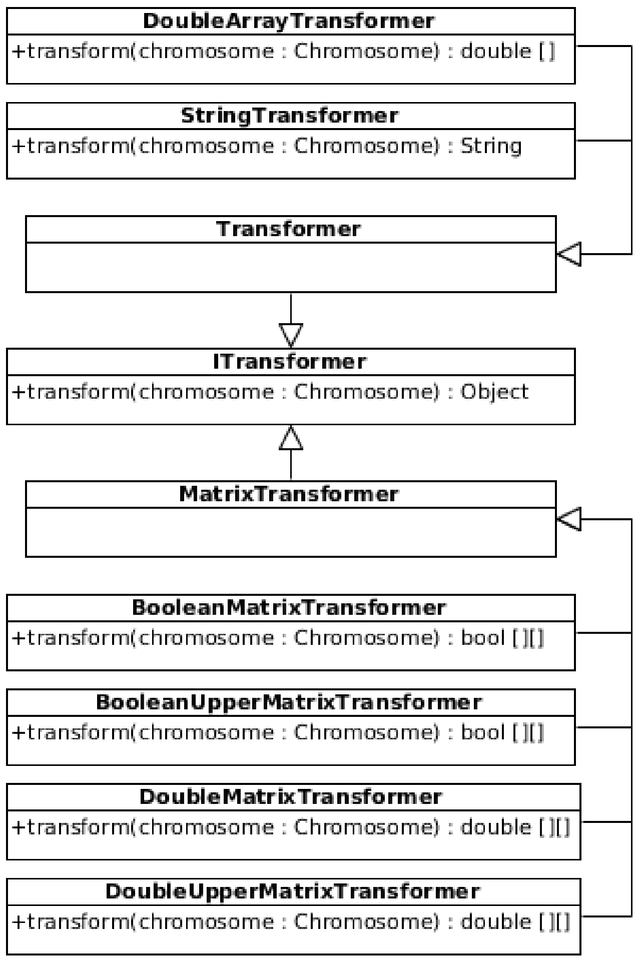

DoubleArrayTransformer
+transform(chromosome : Chromosome) : double [ ]
StringTransformer
+transform(chromosome : Chromosome) : String
Transformer
ITransformer
+transform(chromosome : Chromosome) : Object
MatrixTransformer
BooleanMatrixTransformer
+transform(chromosome : Chromosome) : bool [ ][ ]
BooleanUpperMatrixTransformer
+transform(chromosome : Chromosome) : bool [ ][ ]
DoubleMatrixTransformer
+transform(chromosome : Chromosome) : double [ ][ ]
DoubleUpperMatrixTransformer
+transform(chromosome : Chromosome) : double [ ][ ]

Figure A.9: Transformers

Bibliography

[1] P.J. Angeline, G.M. Saunders, and J.B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *Neural Networks, IEEE Transactions on*, 5(1):54–65, jan 1994.

[2] T. Brázdil. Neuronové sítě. [teaching materials]. [rev. 2011-06-30], [cit. 2012-06-05].

[3] E. Cantu-Paz and C. Kamath. An empirical comparison of combinations of evolutionary algorithms and neural networks for classification problems. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 35(5):915–927, oct 2005.

[4] A. E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag, 2003.

[5] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, March 2003.

[6] J. Heaton. *Introduction to Neural Networks for Java, Second Edition*. Heaton Research, Inc., 2008.

[7] Kyoung jae Kim. Artificial neural networks with evolutionary instance selection for financial forecasting. *Expert Systems with Applications*, 30(3):519 – 526, 2006. <ce:title>Intelligent Information Systems for Financial Engineering</ce:title>.

[8] A.K. Jain, Jianchang Mao, and K.M. Mohiuddin. Artificial neural networks: a tutorial. *Computer*, 29(3):31 –44, mar 1996.

[9] J.R. Koza and J.P. Rice. Genetic generation of both the weights and architecture for a neural network. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume ii, pages 397 –404 vol.2, jul 1991.

[10] V. Landassuri-Moreno, V. Landassuri, and J.A. Bullinaria. Feature selection in evolved artificial neural networks using the evolutionary algorithm epnet, 2010.

[11] Yong Liu and Xin Yao. Evolutionary design of artificial neural networks with different nodes. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 670–675, 1996.

[12] V. Maniezzo. Genetic evolution of the topology and weight distribution of neural networks. *Neural Networks, IEEE Transactions on*, 5(1):39–53, jan 1994.

[13] R. Neruda and J. Šíma. *Teoretické otázky neuronových sítí*. MATFYZPRESS, 1996.

[14] D.T. Pham, M. Castellani, and A.A. Fahmy. Evolutionary feature selection for artificial neural network pattern classifiers. In *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, pages 658 –663, june 2009.

[15] Marylyn Ritchie, Bill White, Joel Parker, Lance Hahn, and Jason Moore. Optimization of neural network architecture using genetic programming improves detection and modeling of gene-gene interactions in studies of human diseases. *BMC Bioinformatics*, 4(1):28, 2003.

[16] R. Rojas. *Neural Networks*. Springer-Verlag, Berlin, 1996.

[17] E. T. Rolls and A. Treves. *Neural networks and brain function*. Oxford University Press, 1998.

[18] Z. Sevarac. Java neural network framework neuroph. [online]. [cit. 2013-09-06].

[19] J. Teda and Z. Lehocký. Genetické algoritmy a jejich aplikace v praxi. [online]. [cit. 2012-05-15].

[20] Xin Yao. Ensemble structure of evolutionary artificial neural networks. *Proceedings of the IEEE*, pages 659–664, may 1996.

[21] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, sep 1999.

[22] Xin Yao and Yong Liu. Making use of population information in evolutionary artificial neural networks. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 28(3):417 –425, jun 1998.

[23] F. Zbořil. Genetické algoritmy. [teaching material]. [rev. 2010-11-30], [cit. 2012-05-15].

[24] Zhi-Hua Zhou, Jianxin Wu, and Wei Tang. Ensembling neural networks: Many could be better than all. *Artificial Intelligence*, 137(1–2):239 – 263, 2002.