

MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY



Tutorial demo of neural networks and genetic algorithms

MASTER THESIS

David Kabáth

Brno, spring 2010

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: Mgr. Radek Pelánek, Ph.D.

Abstract

The thesis aims to introduce a field of *bio-inspired programming*. It focuses mainly on the *neural networks* and *genetic algorithms*. The areas are presented in a form of Java applets that solve given tasks whereon capabilities of aforementioned approaches are easy to see. Neural networks are used for a *language recognition of a text* based on occurrence frequencies of letters. And a task of *Robby, the soda-can-collecting robot* makes use of genetic algorithms. Demonstrated applets are appropriately supplemented with a theoretical background, which is necessary for understanding applications' architecture. At last, results of a few experiments conducted with applets' various settings are presented.

Acknowledgement

I thank my supervisor Radek Pelánek who never refused any of my requests for help and kindly advised me during my work.

Keywords

adaptation, applet, artificial intelligence, artificial neural network, bio-inspired computing, genetic algorithm, Java

Contents

1	Introduction	2
2	Neural networks	3
2.1	<i>Theory</i>	3
2.1.1	Biological inspiration	3
2.1.2	History	5
2.1.3	Fundamentals of an artificial neuron	6
2.1.4	Neural network	10
2.1.5	Perceptron neural networks	10
2.2	<i>Application – language recognition applet</i>	17
2.2.1	Assignment	17
2.2.2	Solution concept	19
2.2.3	Resulting applet	22
2.2.4	Experiments with network’s setup	28
2.2.5	User’s guide	33
3	Genetic algorithms	35
3.1	<i>Theory</i>	35
3.1.1	Biological inspiration	35
3.1.2	History	36
3.1.3	General concept of genetic algorithms	37
3.1.4	Initialization	39
3.1.5	Selection	40
3.1.6	Reproduction	41
3.1.7	Summary	42
3.2	<i>Application – Robby, the soda-can-collection robot</i>	43
3.2.1	Assignment	43
3.2.2	Solution concept	44
3.2.3	Resulting applet	47
3.2.4	Experiments with applet’s setup	53
3.2.5	User’s guide	55
4	Conclusion	57
A	List of Robby’s situations	60
B	Java applet deployment	63
C	Content of the enclosed compact disc	65

Chapter 1

Introduction

In the second half of the last century computer science came up with a new approach. All methods professing this approach, form a group called *bio-inspired computing*. As the term signifies, this group comprises the areas having one common trait – inspiration by nature. A lot of nowadays well-known methods fall within bio-inspired computing. Let's name a few of them:

Neural networks denote parts of one's neural system in neuroscience. Single network is defined by a population of neurons. *Neurons* are the base elements that provide the propagation of excitement through the network. Simply speaking, when fingertips touch a table, the information about what happened, starts to propagate through the body to the effectors. And a transport of the excitement is carried out by means of neurons shaped into neural network.

Artificial neural network is a computational model inspired by its biological pattern. It consists of an interconnected group of *artificial neurons*. The information is distributed through the neurons, nevertheless the main trait is the ability to learn. It is referred to as *adaptation*.

Genetic algorithms represent a set of computational procedures utilizing basics of evolutionary biology. It is a search technique for solving optimization problems. The main idea of this approach is to make use of evolutionary tools such as *inheritance, selection, crossover* and *mutation*.

Members shaping a search space of a problem are encoded as strings of characters. At first, a random population of a given number of individuals is generated. A following generation of the population is evolved by using the evolutionary tools. Subsequently a next generation based on the just evolved one is created in the same way. And the process continues, where every other generation aims to get closer to a solution of the problem.

Let's briefly mention a few more examples: *Cellular automata* are used for artificial life modelling; *Emergent systems* denote complex systems arisen out of relatively simple interactions; field of *swarm intelligence* describes and trades on the collective behaviour of independent agents.

The aim of the thesis is to elucidate two above-mentioned bio-inspired areas. The work itself is divided into two parts. Chapter 2 deals with *Neural networks* and chapter 3 treats of *Genetic algorithms*. Both sections are independent of one another, therefore it is possible to start reading any of them. Each chapter comprises two parts. While the first one is devoted to theory, the second presents a practical task. Solution of the task is represented as a Java applet. Certain theoretical aspects are thereon discussed as well.

Chapter 2

Neural networks

At first starts this chapter with a biological pattern of neural networks, consequently continues with a brief historical summary of developing first artificial models. Then it follows with a division of networks activity into three parts: architecture, active phase, adaptation. Finally models of Perceptron and Multi-layer network are described.

Second part presents an implementation of the neural networks on a simple task, language recognition of the given text. Solution is discussed from the user's and programmer's point of view.

2.1 Theory

2.1.1 Biological inspiration

Elementary results of a research of biological neural networks came up in the second half of the nineteenth century. They were published by a philosopher Herbert Spencer (1820 – 1903), a neuropathologist Theodor Meynert (1833 – 1892) and a well-known neurologist and a psychologist Sigmund Freud (1856 – 1939). They were trying to clarify basic behaviour of a human brain. However, the fundamental idea in a neuroscience called *neuron doctrine* was formally formed by an anatomist Heinrich Wilhelm Gottfried von Waldeyer-Hartz (1836 – 1921). Neuron doctrine stands for the theory that a nervous system is made up of discrete individual cells. The term of *neuron* as the name of an individual cell was coined by Waldeyer as well.

Although the idea of the interconnected neurons forming networks was known in the early twentieth century, the fundamental break-through came with a research of anatomist Santiago Ramón y Cajal (1852 – 1934). He made use of a silver staining method¹ to stain the cells in a nervous tissue. Consequently the neurons became better observable and hence was the way for studying neuron opened.

Outer structure of a neuron can vary in some parameters as a shape or size, nonetheless the inner structure is more or less the same for all (see figure 2.1). A neuron consists of a body called *soma* and its input and output extensions. Input channels, named *dendrites*, are directly connected with a soma. Their number varies depending on a neurons function, nevertheless it can be up to one hundred thousand per neuron. On the contrary, there is just one output called *axon*. However, the end of the axon has got a great number of furcating terminals. This means, that a single *axon terminal* can be considered as a single output, but each of the terminals carries the same information at the moment. That is the opposite to the dendrites, that are able to take over different information to each other.

1. Silver staining method, called Golgi's method, was developed by a physician Camillo Golgi (1843 – 1926). Potassium dichromate and silver nitrate are used to impregnate nervous tissue, in consequence the cell fills by microcrystallization of silver chromate.

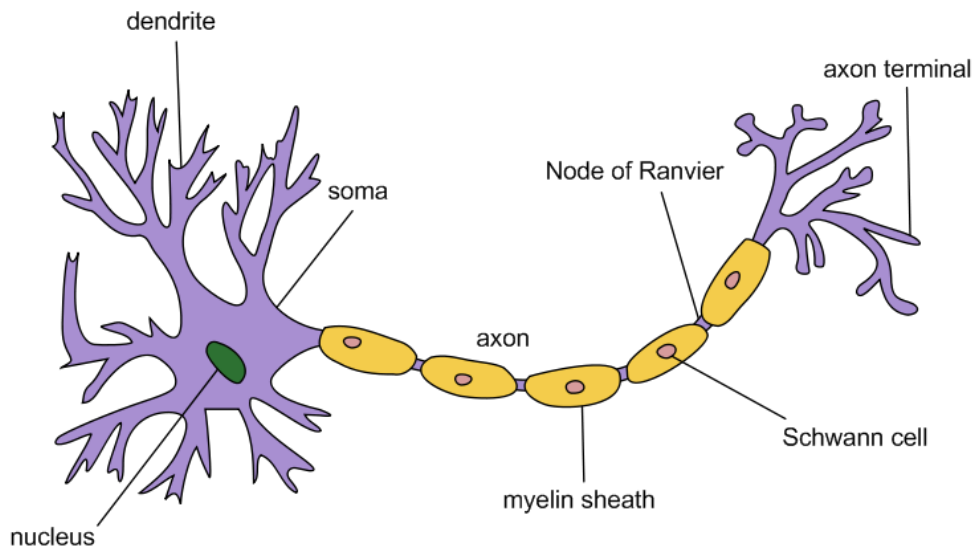


Figure 2.1: Neuron structure, source: <http://en.wikipedia.org/wiki/Neuron>

Let's continue with an inter-neuronal communication, that we started a little bit. The idea of the neural network is that neurons have the ability to transmit the information. Each neuron has a given threshold within. When the threshold is reached or exceeded, the neuron sends the information forward. This process happens in the place named *synapse* (see figure 2.2), where an axon terminal and a dendrite are very close with one another (about 20 nanometers). As we said, when the threshold is reached, a neuron decides to hand its information over. Membrane covering a soma and an axon generates an electrical impulse that starts a chemical process in the synapse. In consequence, a neurotransmitter² is released from an axon terminal into the another neurons dendrite and hereby the information is transmitted.

However, it does not clarify the main function of neural networks – *memory*. Neurons form the memory paths that represent some memorized information. The ability of memorizing is provided by varying permeability of the membrane covering the neuron. When the membrane permeability of the neuron in the certain path increases, the memory path becomes stronger, whereas by decreasing permeability, the memory path weakens or ceases. In this way one learns or forgets.

This is surely not a comprehensive explication, but hopefully it presents the basic ideas of neuron structure and inter-neuronal communication. For more information, we recommend the book called *Biological Neural Networks: Hierarchical Concept Of Brain Functions* written by Konstantin V. Baev[1].

Let's end with Lyall Watson's³ quotation:

"If the brain were so simple we could understand it, we would be so simple we couldn't."

2. Neurotransmitters are chemicals such as acetylcholine or amino acids glycine and adrenaline

3. Lyall Watson (1939-2008) was a South African botanist and zoologist

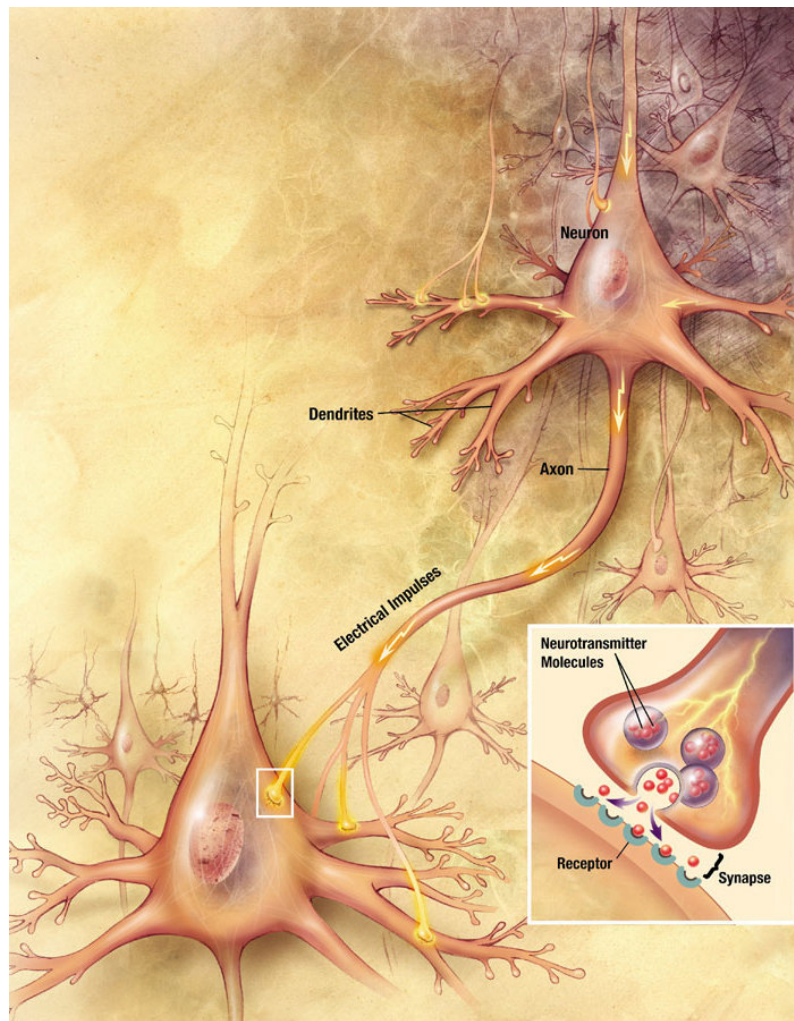


Figure 2.2: Inter-neuronal communication, source: http://en.wikipedia.org/wiki/Chemical_synapse

2.1.2 History

Note: Since now we call the *artificial neural networks* just *neural networks*. The same holds for terms of *artificial neuron* and *neuron* as well. It would not be misleading, because next sections deal mainly with the artificial phrases. If the biological terms are mentioned, it is appropriately emphasized.

A paper written by Warren McCulloch and Walter Pitts[2] in 1943 can be treated as a birth of the artificial neural networks. They brought in the simple mathematical representation of the biological neuron and constructed a primitive neural network based thereon using electrical circuits.

Although the basic model of a neuron was framed, still a proper characterization of a learning process was missing (i.e. description of a biological process of varying membrane permeability – see section 2.1.1). In 1949 wrote Donald Hebb a book called *The organization of behavior* [3], where he pointed out that neural paths are strengthened each time they are used. If two neurons in the path fire together, their connection is amplified. Otherwise it is weakened.

In the next years a few neurocomputers were constructed and put into operation, but in general went unnoticed. Thus continued the theoretical research. In 1957 Frank Rosenblatt developed a new neuron model – *perceptron*[4]. It was an extension of the original model by McCulloch&Pitts to the real numbers. Rosenblatt formulated a new learning rule and mathematically proved its convergence to net configuration solving given task (provided it exists).

Two years later came up Professor Bernard Widrow and his graduate student Ted Hoff with an improved learning rule applied to a new neuron model named *ADALINE*⁴[5]. A network of ADALINEs, called *MADALINE*⁵, is used up to the present day. An example of its application is a system for eliminating echos on phone lines.

Early sixtieth experienced neural networks accruing interest of researchers as well as general public. Unfortunately the interest was one of the causes of gradual decline started a few years later. Initial enthusiasm was a cause of excessive expectations and as time went on, the research progress was slower than expected. The final nail in the coffin was a book *Perceptrons*[6] written by Marvin Minsky and Seymour Papert. They argued that a single perceptron cannot compute XOR logical function and put a question, if the concept of neural networks is worthwhile, when it cannot manage with this simple task. It was already known that this problem solves a two-layered network with three neurons, unfortunately the learning rule for multiple-layered perceptron network was not known. Despite of the fact, that some of ideas in the book were not proved, the criticism were taken over by research community. Next decades are known as a *AI winter*⁶.

Fortunately the eighties came up with a neural network renesance. Thereto largely contributed physicist John Hopfield who interconnected the physics and neural networks in his model commonly known as *Hopfield networks*[7]. Success of his approach benefited mainly from bidirectional connected neurons.

In 1986 published David Rumelhart, Geoffrey Hinton and Ronald Williams[8] a learning algorithm for multi-layer network – *backpropagation*. Thereby they solved the criticism of Minsky and Papert from 1969. Interesting is, that backpropagation was in silent described by Arthur Bryson and Yu-Chi Ho already in 1969[9].

Since then the neural network research was fully reestablished and produced some promising project such as *NETtalk*⁷. Now the research on the field of neural networks continues and the progress lies mainly in the developement of specialized hardware. Near future shows whether it is worthwhile.

2.1.3 Fundamentals of an artificial neuron

Previous two sections elucidated the biological insipiration and historical ground of the neural networks. Therefore we can move forward to artificial neuron scheme and its principles. Consequently, we assign and clarify a few terms, that are helpful to beware of any misleading. Following theory is based on the book of czech scientists Jiří Šíma and Roman Neruda called *Theoretical issues of neural networks*[10].

4. ADALINE – ADAptive LINear Element

5. MADALINE – Multiple ADALINE

6. AI – artificial intelligence

7. *NETtalk* is the project of Terrence Sejnowski and Charles Rosenberg presented in the mid-eighties. It is probably the best-known example of applying backpropagation. Intention of their research was to convert english written text into spoken word

The scheme of *artificial neuron* results from the biological pattern. Let's describe all its parts from the inputs to the output according to figure 2.3.

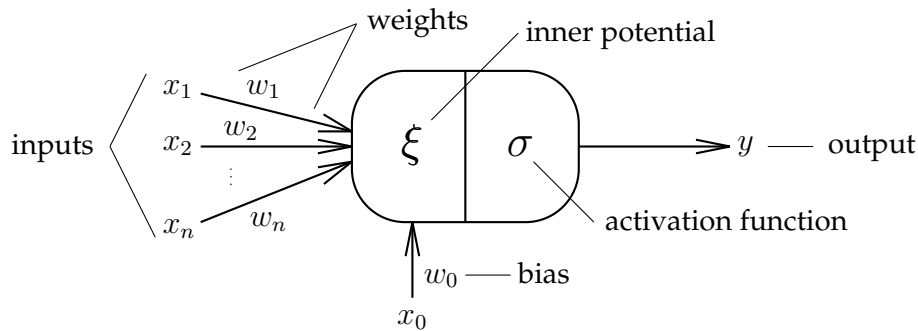


Figure 2.3: Artificial neuron model

- *inputs* – Neuron consists of n inputs representing dendrites of the biological model. We can formally denote them as a vector (x_1, x_2, \dots, x_n) .
- *weights* – Each input is weighted with its synaptic weight. The weight simulates the permeability of the membrane. The bigger the weight is, the more permeable membrane would be in the corresponding biological neuron. Therefore we can write the weights as a vector of n numbers (w_1, w_2, \dots, w_n) .
- *bias* – According to biology, the neuron provides an output when the threshold is reached. Negative value of the threshold t is represented as a weight of a special input in the artificial model called bias. It means $w_0 = -t$. For its formal input x_0 holds that at any time $x_0 = 1$. Hence the value of bias w_0 is fully used when computing an inner potential.
- *inner potential* – The weighted sum of all inputs (including the bias) is called inner potential. Formally:

$$\xi = \sum_{i=0}^n w_i x_i \quad (2.1)$$

- *activation function* – Inner potential is evaluated by an activation function. There are more various functions used in the field of neural networks. When using the most basic one, *unit step function*⁸, the value of the function is (see figure 2.4 for its graph representation):

$$\sigma(\xi) = \begin{cases} 1, & \xi \geq 0 \\ 0, & \xi < 0 \end{cases} \quad (2.2)$$

8. Also called *Heaviside step function*.

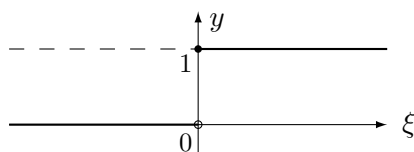


Figure 2.4: Activation function – Unit step function

- *output* – The value of the activation function is denoted by y . It is the output of the neuron:

$$y = \sigma(\xi) \quad (2.3)$$

Let's make a short excursion and demonstrate the function of a bias representing a threshold. According to formula 2.2 it could seem that the neurons threshold equals 0. Nevertheless, the threshold is figured by a bias (as defined above). As consistent with the formula 2.1 and assuming that at any time $x_0 = 1$ holds

$$\xi = w_0 + \sum_{i=1}^n w_i x_i . \quad (2.4)$$

Substituting an inner potential ξ on the right side of the formula 2.2, we can reform it into

$$\sigma(\xi) = \begin{cases} 1, & w_0 + \sum_{i=1}^n w_i x_i \geq 0 \\ 0, & w_0 + \sum_{i=1}^n w_i x_i < 0 \end{cases} \quad (2.5)$$

and consequently in accordance to the definition of bias $w_0 = -t$ where t is a threshold to

$$\sigma(\xi) = \begin{cases} 1, & \sum_{i=1}^n w_i x_i \geq t \\ 0, & \sum_{i=1}^n w_i x_i < t \end{cases} . \quad (2.6)$$

Therefore the neuron outputs 1 when the weighted sum of its inputs x_1, x_2, \dots, x_n reaches or exceeds threshold t and 0 otherwise.

The function of the previous equations could be hard to imagine, hence the mechanisms of a neuron can be more obvious in a graphical interpretation illustrated on the figure 2.5. Let's suppose that for the inner potential in the formula 2.4 holds $\xi = 0$. Hereby we get an equation

$$w_0 + \sum_{i=1}^n w_i x_i = 0 . \quad (2.7)$$

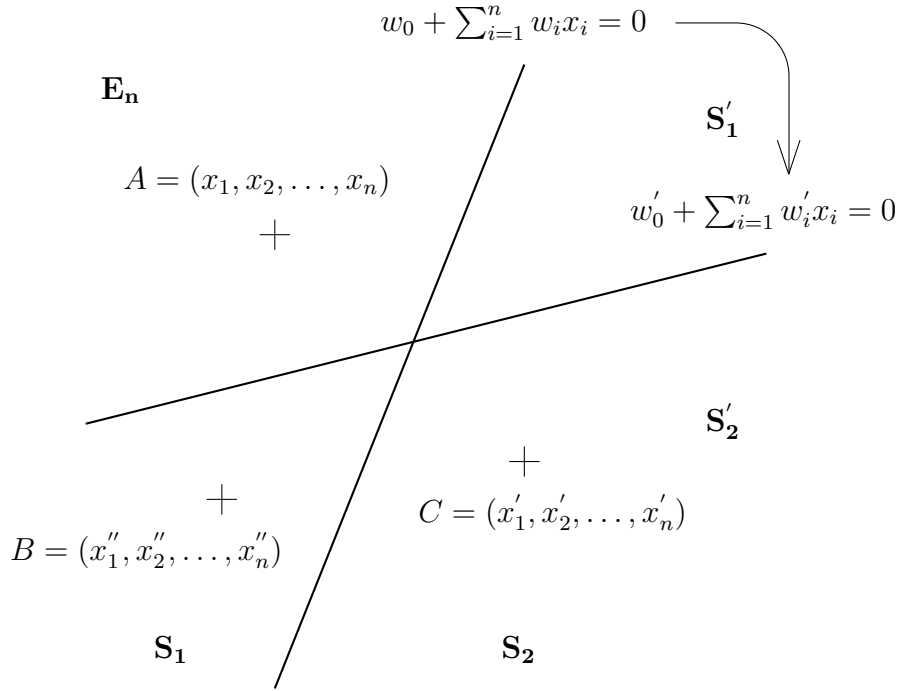


Figure 2.5: Geometric interpretation of the neuron function

This equation determines an object that splits the Euclidean space E of a dimension n into two subspaces S_1 and S_2 . The object varies depending on the dimension of E . It is a line for E of dimension 2 and a plane for dimension 3. Let's call it generally *separator*. Space E contains three points – $A(x_1, x_2, \dots, x_n)$, $B(x'_1, x'_2, \dots, x'_n)$ and $C(x''_1, x''_2, \dots, x''_n)$. These points represent three diverse inputs of a neuron. For the points (A, B as well) involved into the subspace S_1 holds

$$w_0 + \sum_{i=1}^n w_i x_i > 0 \quad (2.8)$$

and on the contrary, points (e.g. C) from the subspace S_2 satisfy

$$w_0 + \sum_{i=1}^n w_i x_i < 0. \quad (2.9)$$

Notice that $\xi = w_0 + \sum_{i=1}^n w_i x_i$. Hence neuron inputs from the subspace S_1 outputs 1 as consistent with formula 2.2, conversely inputs within the subspace S_2 outputs 0. To cover the whole space E of input vectors, we have to add that points determining the separator output 1 (see formula 2.2).

With the help of figure 2.4 we would like to illustrate the most important function of the neuron – *ability of adaptation*. What if the space E is not divided according to our expectations? We can change the coefficients w_0, w_1, \dots, w_n of E . Hereby we get a new coefficients w'_0, w'_1, \dots, w'_n and a separator $w'_0 + \sum_{i=1}^n w'_i x_i = 0$. As illustrated on the figure 2.5, altering the coefficients causes reordering of the subspaces. Point B , originally placed in the subspace S_1 and outputted 1, newly lies in the subspace S'_2 and outputs 0⁹.

9. The whole example illustrated on the figure 2.5 supposes that points of subspace S_1 outputs 1 and points of subspace S_2 outputs 0. Of course, the outputs could be mutually commuted. The assignment of the outputs was just our choice.

2.1.4 Neural network

Neural network denotes a set of interconnected neurons, that influence each other by their computation. Analogous with the biological neural networks (see section 2.1.1), the output of one neuron is the input of other neurons.

For better understanding let's figuratively divide the neural network into three parts:

- *architecture* – represents a structure of the network, how its neurons are connected. It can be simply imagined as a view on the network from outside.
- *active phase* – is an opposite to the architecture. It describes the inwards of the network – what happens from the moment, when the input enters the network till the computations reach its output.
- *adaptation* – is a networks reaction on the ongoing computations. It denotes alterations of the neurons weights.

Let's shortly summarize the whole function of the neural network. At first we structure the neurons to shape some *architecture*. We have certain idea, what the network should do, which task should it fulfil. Consequently, we prepare a set of learning data that fits in the idea. Now we are ready to start the learning process – *adaptation*. When the adaptation is finished, we can start to use the network by inputting the data and letting them compute – to start the *active phase*.

2.1.5 Perceptron neural networks

As we went through the history of neural networks in section 2.1.2, we mentioned a neuron model called *perceptron*. Perceptron is a widely used model up to these days. Therefore we decided to use it in the application presented in section 2.2. Let's define necessary theoretical background at first:

Architecture Perceptrons are used in a *multilayer* structures. Layers are called *input*, *hidden* and *output layer* in turn. Input and output layer is exactly one in contrast with hidden layers. Perceptron neural network can contain unlimited number of hidden layers, nevertheless usually are not used more than two¹⁰. Each layer is connected with just the next one in such way, that inputs of each neuron in the given layer are outputs of all neurons included in the previous layer (see the figure 2.6). Let's just add, that neurons in the input layer have just one input. This architecture is called *feedforward*.

10. Reasons for this low number of hidden layers is time-consuming computations and the fact, that the effect of different hidden layers number has not yet been completely described.

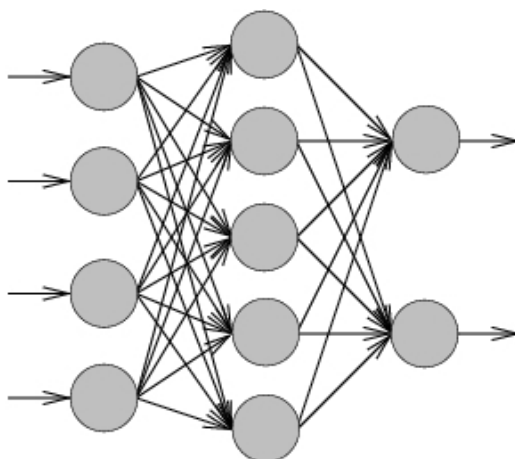
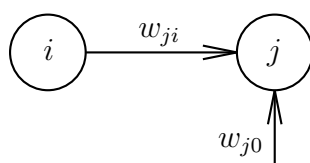


Figure 2.6: Feedforward neural network

We summarize the notation of the neural network:

- Set X includes the neurons in the input layer. Vector $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$ denotes the input of whole network. Values x_1, \dots, x_n respectively represent the input of the neurons in the input layer.
- Set Y contains the neurons in the output layer. Vector $\mathbf{y} = (y_1, \dots, y_m) \in \mathbb{R}^m$ represents the output of the whole network. Values y_1, \dots, y_m are respectively the outputs of the neurons in the output layer.
- Synaptic weight w_{ji} ($i = 1, \dots, n; j = 1, \dots, m$) belongs to the input of neuron j , that represents the output of the neuron i at the same time. Bias of the neuron j is represented as w_{j0} (see figure 2.7). Hence we coupled the weight to the input, we have to complete the definition by adding, that input neuron's weight is always set to 1, but there is no need to deal with them further.
- At last, we define the term *configuration*, that denotes a vector \mathbf{w} of actual values of all weights in the entire network.

Figure 2.7: Synaptic weight w_{ji} connecting two neurons

Active phase

Neural network itself computes the function $f(\mathbf{w}) : \mathbb{R}^n \rightarrow (0, 1)^m$, where \mathbf{w} states for configuration.

Whole computation starts from input layer X . Each neuron j within X has got just one input x_j and bias $w_{j0} = 0$. And by reason of $w_{j1} = 1$ is the inner potential $\xi_j = x_1$. For determination of the output y_j is used the activation function

$$\sigma(\xi) = x \quad (2.10)$$

and because of $y = \sigma(\xi)$, the output of neuron $j \in X$ is $y_j = x_j$. Hence the input neurons can one imagine as a transfer stations, that spread the input further into the network (see the figure 2.6 for getting better idea how the input layer works).

After the computation of the input layer neurons continues the active phase with the contiguous hidden layer. The process is conformable to input layer, only used activation function differs. For each neuron in the given hidden layer is computed inner potential

$$\xi = \sum_{i=0}^l w_i x_i \quad (2.11)$$

where l stands for number of neurons in the previous layer.

Neuron in the hidden layer has got as many inputs as the number of neurons in the previous layer (see figure 2.6). Subsequently the activation function is used (see figure 2.8 for its graph representation):

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}} \quad (2.12)$$

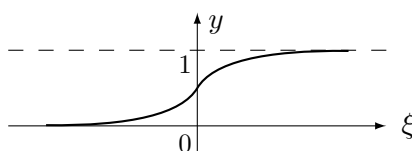


Figure 2.8: Activation function – Standard logistic sigmoid

Seeing that it holds $y = \sigma(\xi)$, the formula 2.12 gives the output of neuron in the hidden layer. The following hidden layers, in turn, are computed in this manner.

Computation of the last (output) layer proceeds the same way as any hidden layer. Number of m neurons included with this layer produce the output of the network $\mathbf{y} = (y_1, \dots, y_m)$.

Adaptation

The most sophisticated part of neural network is the adaptation, that deals with the process of learning. It is realized by altering the weights, or in other words, by changing the configuration of the network.

Adaptation cannot start without knowing the *optimal state*, where no further configuration altering is necessary. Essential fact is, that the *final state* (state after last proceeded active phase during adaptation) mostly doesn't match the optimal state.

The optimal state is given by a set of patterns

$$\mathbf{T} = \left\{ (\mathbf{x}_k, \mathbf{d}_k) \mid \begin{array}{l} \mathbf{x}_k = (x_{k1}, \dots, x_{kn}) \in \mathbb{R}^n \\ \mathbf{d}_k = (d_{k1}, \dots, d_{km}) \in [0, 1]^m \end{array} \quad k = 1, \dots, p \right\} \quad (2.13)$$

where \mathbf{x}_k represents vector of inputs and \mathbf{d}_k vector of desired outputs. Mathematically speaking the aim of the adaptation is to attain during active phase the equation

$$\mathbf{y}(\mathbf{w}, \mathbf{x}_k) = \mathbf{d}_k \quad k = 1, \dots, p. \quad (2.14)$$

To simplify the monitoring of formula 2.14, we define *network error*

$$E(\mathbf{w}) = \sum_{k=1}^p E_k(\mathbf{w}) \quad (2.15)$$

as a sum of errors of single patterns E_k . We define *pattern error* as follows

$$E_k(\mathbf{w}) = \frac{1}{2} \sum_{j \in Y} (y_j(\mathbf{w}, \mathbf{x}_k) - d_{kj})^2 . \quad (2.16)$$

Simply speaking $y(\mathbf{w}, \mathbf{x}_k) - d_k$ is a difference of real output y and desired output d of k th pattern. The optimal state is represented by $E(\mathbf{w}) = 0$, when the vector of real outputs \mathbf{y} equals to the vector of desired outputs \mathbf{d}_k .

The network configuration is altered every time when the set of patterns T is learnt. Consequently the network error $E(\mathbf{w})$ (2.15) is computed and if it is small enough¹¹, the adaptation ends. Otherwise we repeat the process as many time as necessary.

Since the computation of the first pattern's active phase in the set T cannot be accomplished without the values of weights, their initial values are preset. Before the first active phase starts, random number on the interval $[-1, 1]$ assigns to each weight. Therefore, the initial configuration $\mathbf{w}^{(0)}$ consists of random values $[-1, 1]$. Accordingly, we will denote $\mathbf{w}^{(1)}$ the configuration after the first performance of the active phase, $\mathbf{w}^{(2)}$ after the second, $\mathbf{w}^{(3)}$ after the third, etc. We define the configuration $\mathbf{w}^{(t)}$ where $t > 0$ as

$$w_{ji}^{(t)} = w_{ji}^{(t-1)} + \Delta w_{ji}^{(t)} . \quad (2.17)$$

As we can see, the endeavour to reach the optimal state is incremental. Because of our try to minimize the network error $E(\mathbf{w})$, we can use any of well-known optimization methods (e.g. Newton's method¹²). For our purposes we chose gradient method where the increment is defined as follows

$$\Delta w_{ji}^{(t)} = -\varepsilon \frac{\partial E}{\partial w_{ji}}(\mathbf{w}^{(t-1)}) \quad (2.18)$$

and $0 < \varepsilon < 1$ stands for *learning rate*. As well as other optimization methods leads us this gradient method to the local minimum of the function E by altering the weights – see the figure 2.9 for an example. Properties of E and dependence on ε will be discussed later in the section 2.2.

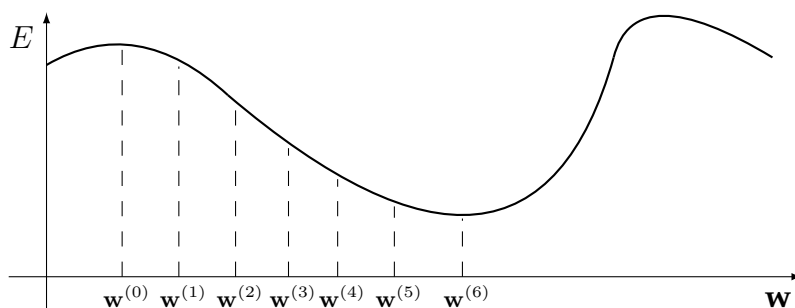


Figure 2.9: Behaviour of the error value regarding to the altering of the configuration

11. "Small enough" is not exact mathematical term, but it fits here the best, because it is just up to us which value it will represent.

12. http://en.wikipedia.org/wiki/Newton_method_in_optimization

Unfortunately the computation of the gradient 2.18 is not trivial and the first solution of the problem was published as late as 1986 (see section 2.1.2). The algorithm is called *backpropagation* and as the name implies the alterations of the neurons' weights proceed from the output layer through the hidden layers to the input layer.

At first we use the addition rule for derivatives

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^p \frac{\partial E_k}{\partial w_{ji}} \quad (2.19)$$

which says that $[g(x) + f(x)]' = g'(x) + f'(x)$. Subsequently we use the chain rule

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \frac{\partial y_j}{\partial \xi_j} \frac{\partial \xi_j}{\partial w_{ji}} \quad (2.20)$$

defined as a $[f(g(x))]' = f'(g(x))g'(x)$. The idea of this step will be probably more obvious when writing the chain rule for derivatives in Leibniz's notation, $\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}$ ¹³. At this time we fragmentated the derivative of the function E into three parts that we can simplify by appropriate modifications and solve. Let's begin with the most right one

$$\frac{\partial \xi_j}{\partial w_{ji}} = \frac{\partial(\sum_{k=0}^n w_{jk}y_k)}{\partial w_{ji}} = y_i \quad (2.21)$$

where derivatives of all elements of the sum defining inner potential ξ_j except one where $k = i$, are equal to 0¹⁴.

For the second part of 2.20 we simply substitute formula 2.12 and differentiate

$$\frac{\partial y_j}{\partial \xi_j} = \frac{\partial(\frac{1}{1+e^{-\xi_j}})}{\partial \xi_j} = \frac{e^{-\xi}}{(1+e^{-\xi})^2} = \frac{1}{1+e^{-\xi_j}}(1 - \frac{1}{1+e^{-\xi_j}}) = y_j(1-y_j) . \quad (2.22)$$

After that we substitute formulae 2.21 and 2.22 into 2.20

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} y_j(1-y_j)y_i . \quad (2.23)$$

To compute the remaining part of 2.20 we use the method that gave name to the whole algorithm – backpropagation. We start with the derivative of neurons in the output layer Y that are easy to differentiate by using the standard rules for derivatives. Consequently we continue in turn with derivatives of neurons in hidden layers from output layer towards the input layer where the values from previous layer are used. Hence the derivative of an error is "back-propagated".

For every neuron $j \in Y$ we differentiate it by using formula 2.16

$$\frac{\partial E_k}{\partial y_j} = \frac{\partial(\frac{1}{2} \sum_{j \in Y} (y_j - d_{kj})^2)}{\partial y_j} = y_j - d_{kj} . \quad (2.24)$$

We used analogous method as in differentiating 2.21¹⁵.

13. Supposing that variable y depends on a variable u which in turn depends on a variable x .

14. $\frac{\partial(\sum_{k=0}^n w_{jk}y_k)}{\partial w_{ji}} = \frac{\partial(w_{j0}y_0 + w_{j1}y_1 + \dots + w_{ji}y_i + \dots + w_{jn}y_n)}{\partial w_{ji}} = 0 + 0 + \dots + y_i + \dots + 0 = y_i$

15. In more detail: $\frac{\partial(\frac{1}{2} \sum_{j \in Y} (y_j - d_{kj})^2)}{\partial y_j} = \frac{\partial(\frac{1}{2} [(y_0 - d_{k0})^2 + (y_1 - d_{k1})^2 + \dots + (y_j - d_{kj})^2 + \dots])}{\partial y_j} = \frac{1}{2} \cdot 2(y_j - d_{kj}) = y_j - d_{kj}$

At last remains the derivative for neuron $j \notin X \cup Y$ – with the help of the chain rule:

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in J} \frac{\partial E_k}{\partial y_r} \frac{\partial y_r}{\partial \xi_r} \frac{\partial \xi_r}{\partial y_j} = \sum_{r \in J} \frac{\partial E_k}{\partial y_r} (1 - y_r) w_{rj} \quad j \notin X \cup Y. \quad (2.25)$$

The formula 2.25 substitutes the partial derivative $\frac{\partial E_k}{\partial y_j}$ by the computation of partial derivations of neurons $r \in J$. Set J represents the neurons, such that one of their inputs is an output of the neuron j . It means that the set J includes the neurons from the following layer of neuron j when supposing the direction from input to output layer (see the figure 2.10).

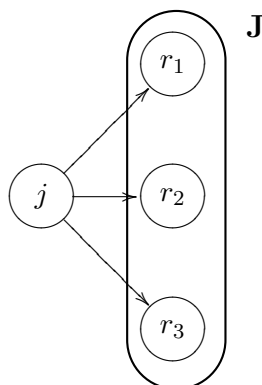
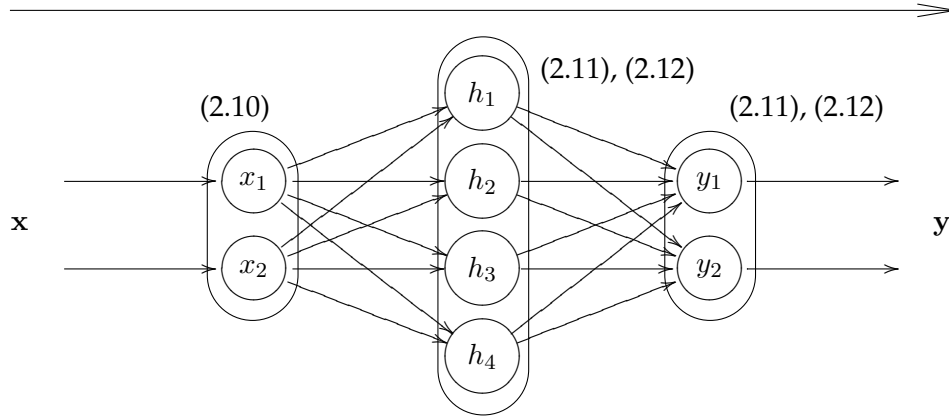


Figure 2.10: Example of the set J

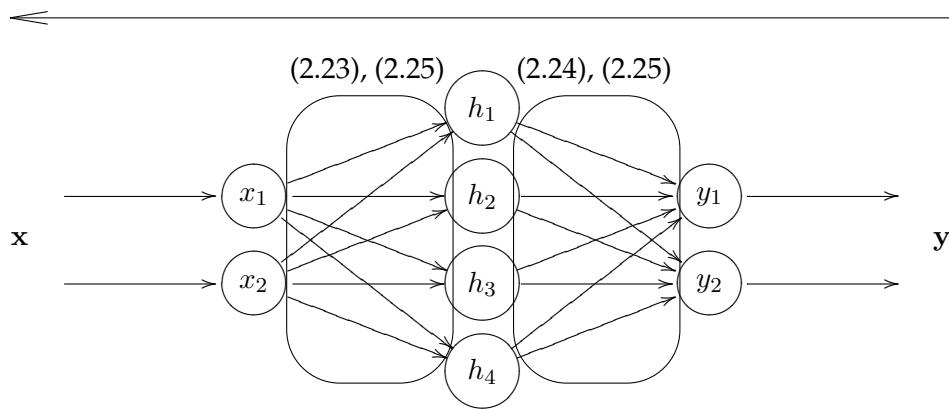
Partial derivations $\frac{\partial \xi_r}{\partial y_j}$ and $\frac{\partial y_r}{\partial \xi_r}$ are computed similarly to the formulae 2.21 and 2.22 (respectively). Finally, when computing the partial derivation $\frac{\partial E_k}{\partial y_r}$, we start with neurons in the output layer, where we use the formula 2.24 and consequently proceed through the hidden layers by using 2.25 in the direction of input layer. In conclusion, let's summarize the process of adaptation – as an algorithm in the box below and graphically on the figure 2.11.

1. Prepare a set of patterns T . Each pattern is a pair (\mathbf{x}, \mathbf{d}) .
2. Randomly generate initial configuration $\mathbf{w}^{(0)}$, where a weight $w_{ij} \in [-1, 1]$.
3. Set an error counter $E_{ji} = 0$ for every weight w_{ji} . E_{ji} represents an error after one learning cycle of a set T .
4. For every pattern (\mathbf{x}, \mathbf{d})
 - (a) compute its output \mathbf{y} according to formulae 2.10 – 2.12.
 - (b) calculate $\frac{\partial E}{\partial w_{ji}}$ for every weight w_{ji} with the help of formulae 2.23 – 2.25.
 - (c) add the current $\frac{\partial E}{\partial w_{ji}}$ to the each weight's error counter E_{ji} .
5. Set new weights for every w_{ji} according to formula 2.17 and 2.18. Substitute $\frac{\partial E}{\partial w_{ji}}$ by E_{ji} , because the error counter for given weight holds the sum of values $\frac{\partial E}{\partial w_{ji}}$ for every pattern.
6. Calculate net error E with help of formulae 2.15 and 2.16.
7. If the net error is not small enough, continue with step 3. Otherwise end.

Prepare the adaptation process by steps 1 – 3 from the box on the previous page.
 For every pattern go through the step 4. At first calculate the output according to substep 4a:

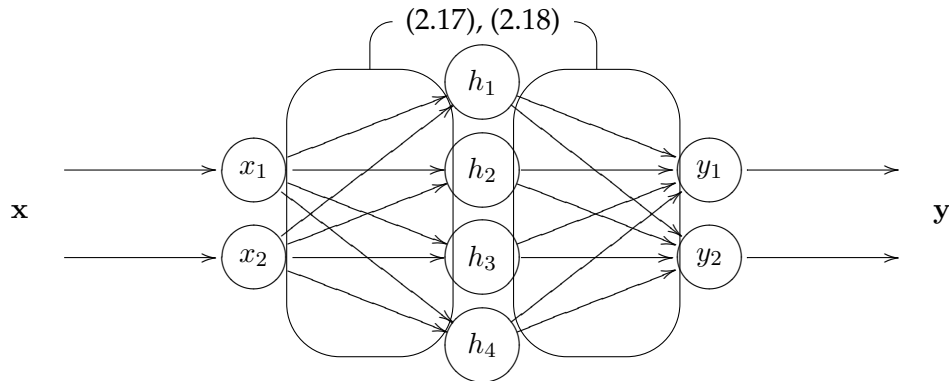


Subsequently continue with substep 4b – calculate $\frac{\partial E}{\partial w_{ji}}$ for every weight starting from inputs of output layer and proceed in the direction of input layer:



In conformance with step 4c add the current value of $\frac{\partial E}{\partial w_{ji}}$ of each weight w_{ji} to corresponding error counter E_{ji} .

When a learning cycle consisting of all training patterns is finished, continue with the step 5. Add corresponding values of E_{ji} to all weights w_{ji} . It holds that $\frac{\partial E}{\partial w_{ji}} = E_{ji}$.



Compute a current value of the net error E . If it is not small enough, continue with a next learning cycle. Otherwise end (step 6 and 7 – formulae 2.15 and 2.16).

Figure 2.11: Adaptation process – step by step

2.2 Application – language recognition applet

Second part of chapter 2 deals with an application of neural networks. It presents a real-world task – an applet recognizing a language of a given text – to demonstrate their usage. At first we define the assignment and slightly outline the concept of the solution. Subsequently we present the applet, go through its parts and describe their structures. Next to the last section discusses the results of experimenting with the applet's setup. We try to show the potential of neural network regarding to its parameters in this section. Final part sums up how to use the applet in a brief User's guide.

2.2.1 Assignment

We tried to choose the assignment to be illustrative enough and easy to understand. Therefore we chose the topic of language recognition of given text based on relative occurrence frequencies of letters.

We have a vector of 26 values representing the relative letter frequencies occurred in a given text. Letters of the english alphabet (A, B, \dots, Z) are used. The vector can be entered manually or as a text file. If the second option is chosen, the applet computes the input vector itself. The process of language recognition will be provided by neural network. As a set of patterns will be used the relative letter frequencies of ten languages – Dutch, English, French, German, Italian, Polish, Portuguese, Spanish, Swedish, Turkish (see the table 2.1 on the page 18). User should be able to control the teaching process by setting up some parameters of neural network. Animation, illustrating the teaching and recognition process of network, will be part of the applet. It would help a user to follow up the work of the applet.

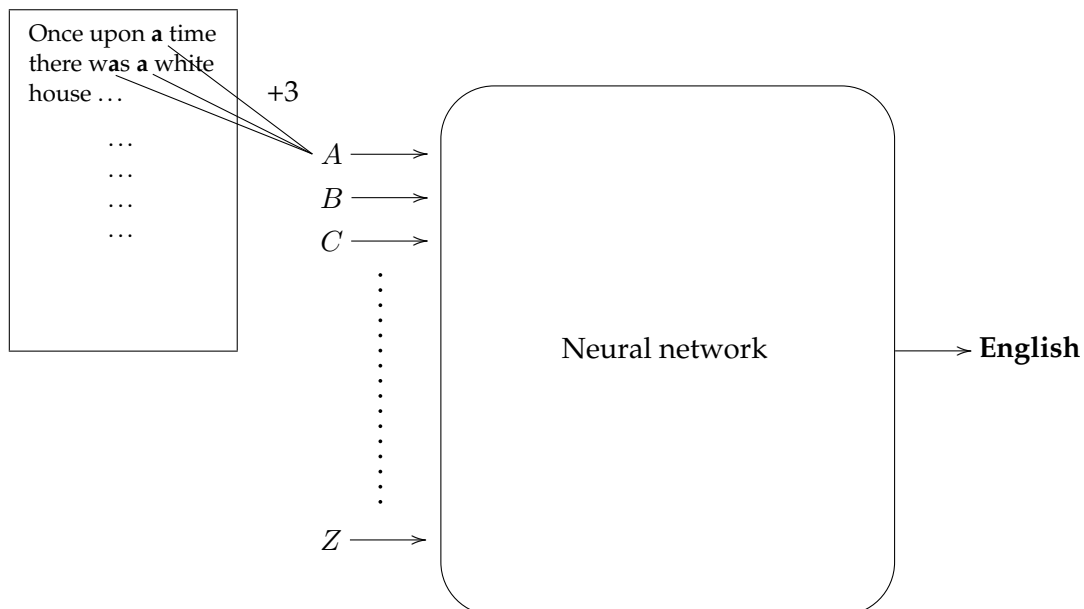


Figure 2.12: Concept of the applet

Letter	Dutch	English	French	German	Italian	Polish	Portuguese	Spanish	Swedish	Turkish
A	7.490	8.167	7.636	6.510	11.740	8.000	14.630	12.530	9.300	11.680
B	1.580	1.492	0.901	1.890	0.920	1.300	1.040	1.420	1.300	2.950
C	1.240	2.782	3.260	3.060	4.50	3.800	3.880	4.680	1.300	0.970
D	5.930	4.253	3.669	5.080	3.730	3.000	4.990	5.860	4.500	4.870
E	18.910	12.702	14.715	17.400	11.790	6.900	12.570	13.680	9.900	9.010
F	0.810	2.228	1.066	1.660	0.950	0.100	1.020	0.690	2.000	0.440
G	3.400	2.015	0.866	3.010	1.640	1.000	1.300	1.010	3.300	1.340
H	2.380	6.094	0.737	4.760	1.540	1.000	1.280	0.700	2.100	1.140
I	6.500	6.966	7.529	7.550	11.280	7.000	6.180	6.250	5.100	8.270
J	1.460	0.153	0.545	0.270	0.000	1.900	0.400	0.440	0.700	0.010
K	2.250	0.772	0.049	1.210	0.000	2.700	0.020	0.010	3.200	4.710
L	3.570	4.025	5.456	3.440	6.510	3.100	2.780	4.970	5.200	5.750
M	2.210	2.406	2.968	2.530	2.510	2.400	4.740	3.150	3.500	3.740
N	10.030	6.749	7.095	9.780	6.880	4.700	5.050	6.710	8.800	7.230
O	6.060	7.507	5.378	2.510	9.830	7.100	10.730	8.680	4.100	2.450
P	1.570	1.929	3.021	0.790	3.050	2.400	2.520	2.510	1.700	0.790
Q	0.009	0.095	1.362	0.020	0.510	0.000	1.200	0.880	0.007	0.000
R	6.410	5.987	6.553	7.000	6.370	3.500	6.530	6.710	8.300	6.950
S	3.730	6.327	7.948	7.270	4.980	3.800	7.810	7.980	6.300	2.950
T	6.790	9.056	7.244	6.150	5.620	2.400	4.740	4.630	8.700	3.090
U	1.990	2.758	6.311	4.350	3.010	1.800	4.630	3.930	1.800	3.430
V	2.850	0.978	1.628	0.670	2.100	0.000	1.670	0.900	2.400	0.980
W	1.520	2.360	0.114	1.890	0.000	3.600	0.010	0.020	0.030	0.000
X	0.040	0.150	0.387	0.030	0.000	0.000	0.210	0.220	0.100	0.000
Y	0.035	1.974	0.308	0.040	0.000	3.200	0.010	0.900	0.600	3.370
Z	1.390	0.074	0.136	1.130	0.490	5.100	0.470	0.520	0.020	1.500

Table 2.1: Relative frequencies of letters' occurrence – percentage unit used

2.2.2 Solution concept

Before programming itself, we had to think out some fundamental problems and prepare the concept of the applet. At first, we decided to use Java as a programming language for its passable visualization features. Therefore the resulting application will be the *Java applet*.

Figure 2.12 from the page 17 shows the concept of the applet. It outlines two separate tasks. An easier one – to parse the given text file – and a more difficult one – to prepare the concept of the neural network that is figured by a black box without knowing its inwards.

Text file parser

As mentioned in the assignment, the relative frequencies of letters in the text can be entered manually or automatically calculated from a given file. The first choice is trivial, the second relatively as well. There just has to be decided in which way single letters will be processed.

We decided to work with ten languages mentioned in the table 2.1 by reason of availability of the data. We simply used the relative frequencies of letters' occurrence in the text presented on the Wikipedia¹⁶. One can argue that our data source is not fully reliable, but we think that today's Wikipedia provides relatively accurate data without emphasis on the values far beyond a decimal point at most. And that is sufficient for the applet because its main task is not to build up a new multi-purpose translator, but only to illustrate the usage of neural networks.

Unfortunately all languages contained in the table 2.1 do not use just 26 letters of the english alphabet (A, B, \dots, Z). Nevertheless we have to work just with them, because we need the set of letters included in all languages that we work with. Hence we divide the text characters into three groups:

- *Letters of english alphabet* – The letters $a, b, \dots, z, A, B, \dots, Z$ belong here. It holds that a is the same letter as A , b as B , etc. We count the number of each of them. Let's imagine it as an array of the size 26. Each of its fields holds the number of single letter.
- *Another countable characters* – National characters and other characters such as $;$, $!$, $?$, etc. We define this set as characters that do not belong to the first neither the third set. We count the total number of them, but do not distinguish between single kinds.
- *Uncountable characters* – Control characters (also called non-printing characters) are the last set. In addition white space and break belongs to this set. We do not count them.

After parsing the text file character by character, we calculate the relative frequency of occurrence for each letter

$$letter\ rel_freq = \frac{|letter\ of\ english\ alphabet|}{|Letters\ of\ english\ alphabet| + |Another\ countable\ characters|} \cdot 100 \quad (2.26)$$

where *letter of english alphabet* stands for a single field of *Letters of english alphabet* array. This way we get a percentage value for each letter of the english alphabet. Deeper discussion of programming the text parser is presented in the Resulting applet section (2.2.3).

16. http://en.wikipedia.org/wiki/Letter_frequency

Neural network concept

The most important task before programming is to decide how the neural network will be shaped. We decided to use *Perceptron multilayer neural network without cycles – feedforward model*. It is completely based on the section 2.1.4 and 2.1.5. If you are not familiar with it, please start with these sections, because if necessary, we just refer to the theory described in the whole section 2.1 from now. We expect better readability of not interlacing the practical task description by equations. Therefore we decided to separate theory from the rest.

At first, we choose the number of layers. We let us inspire with the *NETtalk project* intended to convert an english written text into a spoken word. They used three layers model – one input layer, one hidden layer and one output layer and developed wellworking system. Hence we chose the same three layers architecture.

Consequently we have to assess the number of neurons in each layer. If we have a look at the above-mentioned NETtalk, we find out that they used 203 input neurons, 80 neurons in the hidden layer and 26 output neurons. These numbers are not random. 203 input units correspond to 7 sets of 29 neurons. Each of these sets represents one letter, because the NETtalk was capable to work with words of length 7 at most. Number of 29 symbolizes 26 letters of the english alphabet (A, B, \dots, Z) and *a dot, a comma* and *a space*. There was just one neuron of the set active at the moment (exactly the one representing required character). 26 output units symbolize the english phonemes. And finally the number of neurons in the hidden layer was assessed heuristically.

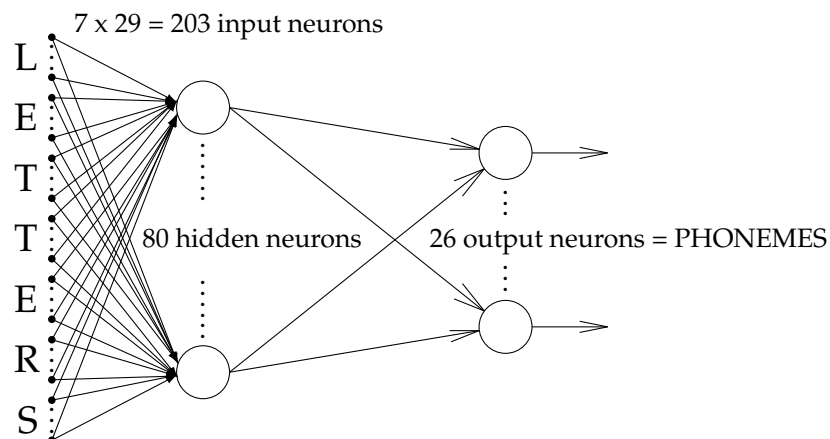


Figure 2.13: NETtalk

But let's go back to our task. We have to assess the numbers of neurons in the layers:

- *Input layer* – By reason of using the english alphabet that contains 26 letters, we decided to use 26 input neurons. Thus each neuron represents one letter. More precisely, input neuron holds the relative frequency of corresponding letter in percentage.
- *Hidden layer* – It is relatively difficult task to assess the most suitable number of neurons in this layer. We drew inspiration from the NETtalk and general practice. It always depends on the current task, but it is recommended to use a few more units than in the input layer. How uncertain this recommendation can be, shows NETtalk that doesn't adhere it at all and works relatively well.

Therefore we left this layer's neuron number as a parameter by which user can influence the capability of the network. Default value is set to 40.

- *Output layer* – Compared to the previous layer, the output layer contains exactly 10 neurons. This value conforms to the number of languages, that we want to recognize. The neuron (and a corresponding language) holding the highest value is declared the winner.

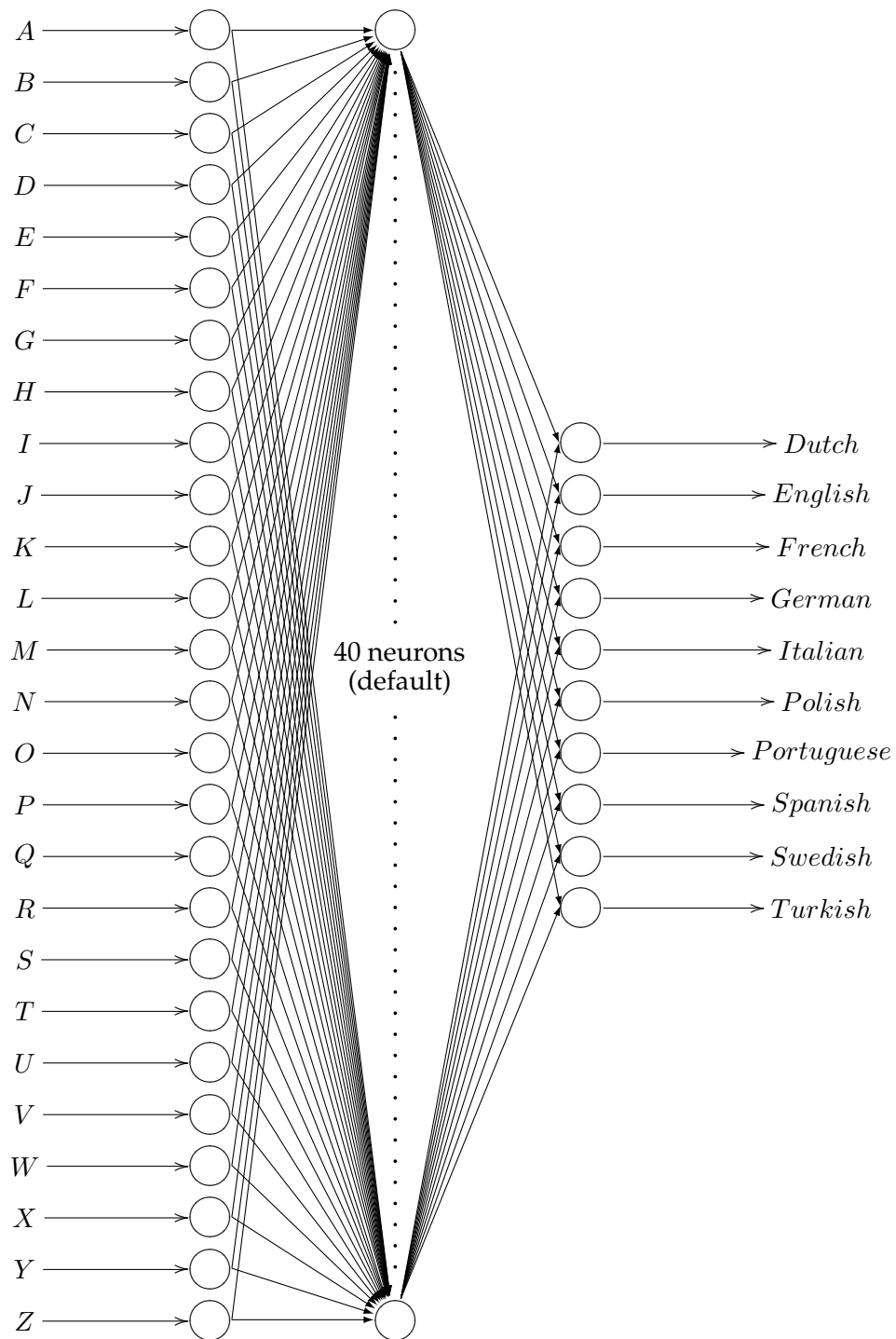


Figure 2.14: Language recognition – concept of the neural network

2.2.3 Resulting applet

After preparing the concept, we should continue with giving an account of the programming phase. Nevertheless, we suppose that it is more beneficial to switch to the top-down strategy and start with the presentation of the entire resulting Java applet. Consequently we divide it into single parts, when each of them solves accurately bounded subtask.

Since the following section deals with applet's functionality in depth, we recommend to continue with the section 2.2.5 named User's guide, if you just want to try it out from the user's point of view.

Applet as a whole

Let's start with the whole applet at first (see the figure 2.17 on the page 23). It is divided into two parts of almost the same size. The left one serves as a *control panel*, where a user regulates the whole functionality of the applet. In contrast, the right one is more or less passive and works as a *visualization display*.

Control panel itself can be subdivided into six smaller sections. We present them one by one on the following pages. Their names can sometimes look awkwardly, but we decided to leave their titles the same as they stay in the applet for greater clarity. In addition, we discuss the functionality of the visualization display in the end.

Status

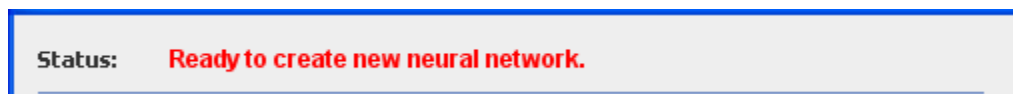


Figure 2.15: "Status" section

This is the least complicated component of the applet. It just serves to inform about the applet's status. It also displays error messages, if occur.

Create new net

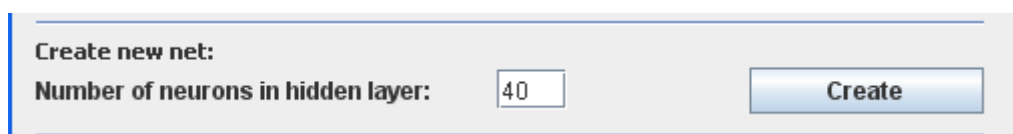


Figure 2.16: "Create new net" section

The first user serviceable component serves for creation of the new network. By clicking the *Create* button, the neural network with specified number of the neurons in the hidden layer is generated. Default value is set to 40. Allowed range contains any integer from the interval (0, 65536). The upper bound of the restriction is appointed due to the capacity of the used data types and conversions between them. On the other hand, this is just the technical boundary, the logical value¹⁷ is not greater than 100 neurons. Unfortunately we are not able to support these statements by any proofs. This value, that is still quite exaggerated, is based

17. Value that makes sense to use to obtain various, but still correct results.

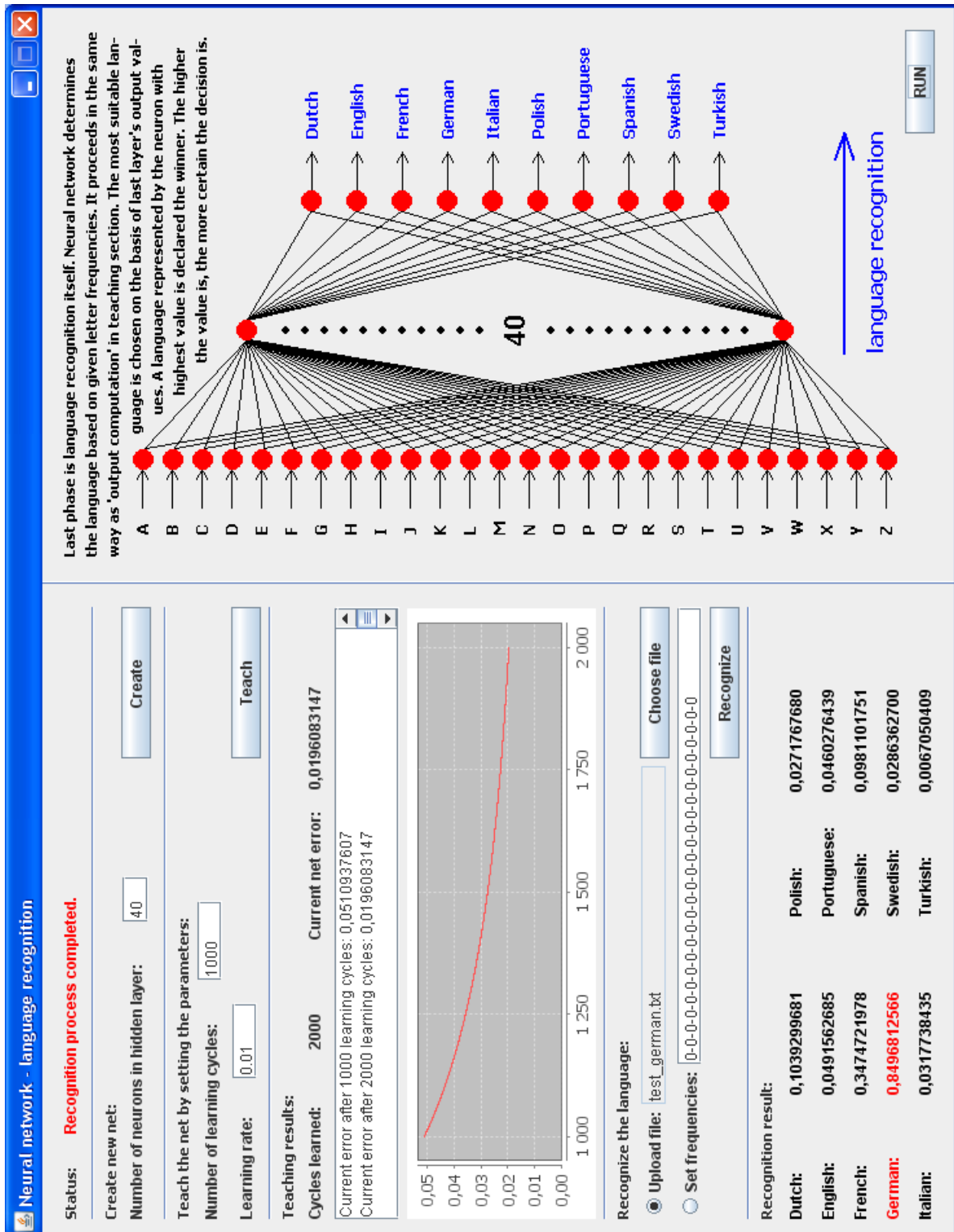


Figure 2.17: Language recognition – resulting Java Applet

on the consideration of the task complexity, general practice and testing procedures (see the section 2.2.4 for their results). Let's add, that the creation process includes an upload of the patterns (the relative letter frequencies from the table 2.1 on the page 18), generating random weights of all synapses from the interval $[-1, 1]$ and computing the current net error. The network is now fully prepared for learning the patterns.

User can also watch the animation that starts in the visualization display and changes when clicking other buttons.

Teach the net by setting the parameters

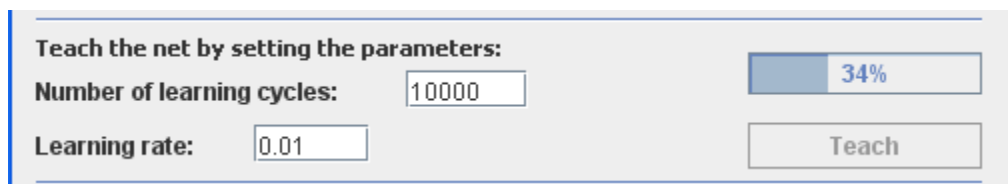


Figure 2.18: "Teach the net by setting the parameters" section

Following step after creating the network is to teach it the patterns. It fully conforms to the theory described in the section 2.1.5 about adaptation. If you are interested in the mathematical background, please switch to this section. We just continue with describing the applet's characteristics herein and referencing to the theory section if necessary.

This component offers the user another possibilities to influence the applet's functionality by another two parameters. The first one defines a *number of learning cycles* – how many times the set of the patterns should be taught. The value has to be an integer greater than 0 and lower than 2 147 483 647. The upper bound is restricted by the capacity of the Java *int* data type. It is preset to 10 000. The second parameter is a *learning rate*. The learning rate represents the parameter ε in the equation 2.18 on the page 18. Simply speaking, it controls the size of the weights' changes after each learning cycle. The parameter's range is an interval of the real numbers $(0, 1)$. The higher the value is, the bigger impact the change causes. It is preset to 0.01.

Although the parameters serve for testing the optimal configuration, we strongly recommend not to use the high value of learning cycles at once. The speed of the learning process naturally depends on the performance of used hardware and the combination of the other parameters (number of neurons in the hidden layer and learning rate). It seems to be better to split the learning process into more parts, when each of them amounts to 100 000 cycles at most.

After clicking the *Teach* button, the learning process of the network starts. Functions of other applet's buttons are disabled to ensure the stability. The learning procedure takes place in the new thread by using the Java class *SwingWorker*¹⁸, that provides required functionality. It serves to processing the long running task in the background and provides the possibility to hand over intermediate results to the applet's main thread. Therefore the applet is able to show the progress bar, when the learning process is on the run. At the end of the learning procedure, other applet's buttons are re-enabled. Let's just add, that this component illustrates its functionality in the visualization display as well as the Create component.

18. See the Java API documentation for more information:

<http://java.sun.com/javase/6/docs/api/javax/swing/SwingWorker.html>

Teaching results

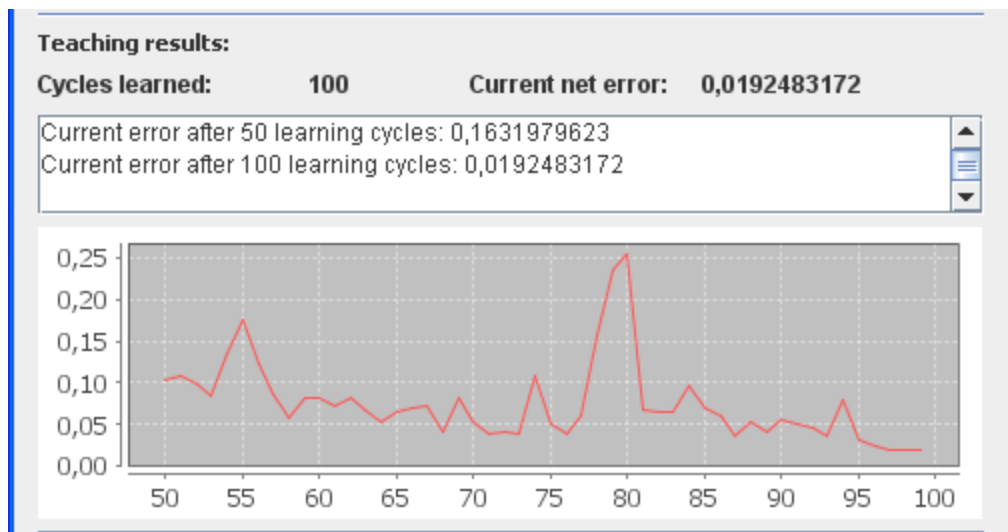


Figure 2.19: "Teaching results" section

It is important to know the results of the teaching process. This function is fulfilled by passive component called "Teaching results". It provides a few important statistics of the currently loaded network. At first, a *number of learned cycles* and *current net error* are displayed here. After that follows the text area, that holds the information about the network gathered during whole teaching process. It is stored and updated here until a new network is created. The largest area of this component is filled by a graph, that displays the current net error with respect to number of learned cycles. All of these statistics are updated after every teaching process. In other words, everytime when the Teach button is clicked and consequently the other buttons are re-enabled, the data is updated.

Let's go back to the plotting tool for a while. We used Java chart library called *JFreeChart, version 1.0.13*¹⁹. This useful utility provides wide range of classes for plotting various graphs. It is downloadable under the LGPL²⁰ licence, that means that it is free to use. We intended to plot the graph online during the teaching process, unfortunately it wasn't possible due to high performance requirements. Therefore we had to put up with plotting the graph just once at the end of every teaching process.

Recognize the language

Moving to the next component we get to the core of the applet (see the following page for the figure). It provides the upload of letter occurrence frequencies and a language recognition procedure itself.

At first, let's follow the concept of the text file parser outlined in the section 2.2.2. In accordance with it, there are two options of uploading the letter frequencies. The first is a text field, where user can manually insert the sequence of 26 real numbers greater than or equal to 0. The compulsory data format is: A frequency–B frequency–...–Z frequency.

19. See the project's homepage <http://www.jfree.org/jfreechart/> for a free download.

20. LGPL = GNU Lesser General Public License

Figure 2.20: "Recognize the language" section

The second option is to upload the text file, whereon the relative frequencies are computed. Because of the miscellaneous extensions of files that can contain the text, we do not constrain their upload. It means, that every file is processed. We did not decide for this solution by the reason of our laziness, but because not to discriminate different text file extensions that are used by diverse operating systems. Since the files are processed character by character, it is up to the user's judgement, which files to upload. All of them are processed without error messages, but some of them will output nonsensical results after language recognition procedure (e.g. image files – jpeg, tiff, ...).

Let's attend to the processing of the uploaded file now. The parsing procedure fully follows the concept specified in the section 2.2.2, hence we just want to add how exactly it is computationally realized. We use the ASCII codes²¹ to distinguish to which group the current character belongs:

Letters of english alphabet		Another countable characters		Uncountable characters	
Letter	ASCII code	Letter	ASCII code	Letter	ASCII code
A	65			0	(Null)
B	66			1	(Start of heading)
⋮	⋮			⋮	⋮
Z	90		All the ASCII codes, that do not belong to the first neither the third set	30	(Record separator)
a	97			31	(Unit separator)
b	98			32	Space
⋮	⋮			127	Delete
z	122			160	Non-breaking space

Table 2.2: Division of the letters into three groups based on ASCII codes

Consequently, we compute the letters' relative frequencies of the occurrence with the help of the formula 2.26 on the page 19 and get the resulting vector.

Afterwards, we can move forward to the language recognition process itself. The active phase characterized in the theoretical section 2.1.5 starts by clicking the Recognize button. Its output is a vector of ten real numbers from the interval (0, 1) that each represent one language. The highest value is declared the winner. The higher the value is, the more probable the selection of the winner was correct.

It is again possible to watch the animation of the recognition process in the visualization display.

21. *The American Standard Code for Information Interchange* (ASCII) is a character-encoding scheme based on the ordering of the English alphabet. (source: <http://www.wikipedia.org>)

Recognition result

Recognition result:			
Dutch:	0,0174199667	Polish:	0,0045553361
English:	0,6803509806	Portuguese:	0,0027444125
French:	0,0066798023	Spanish:	0,0282037184
German:	0,0334806447	Swedish:	0,0198553102
Italian:	0,0429912691	Turkish:	0,0025967868

Figure 2.21: "Recognition result" section

The last component of the control panel is a recognition results display. When the computation provided by previous component is finished, the results are figured here. They are displayed with an accuracy of ten decimal places and the highest value is highlighted by a red color. If a new net is created, the previously calculated results are discarded.

Visualization display

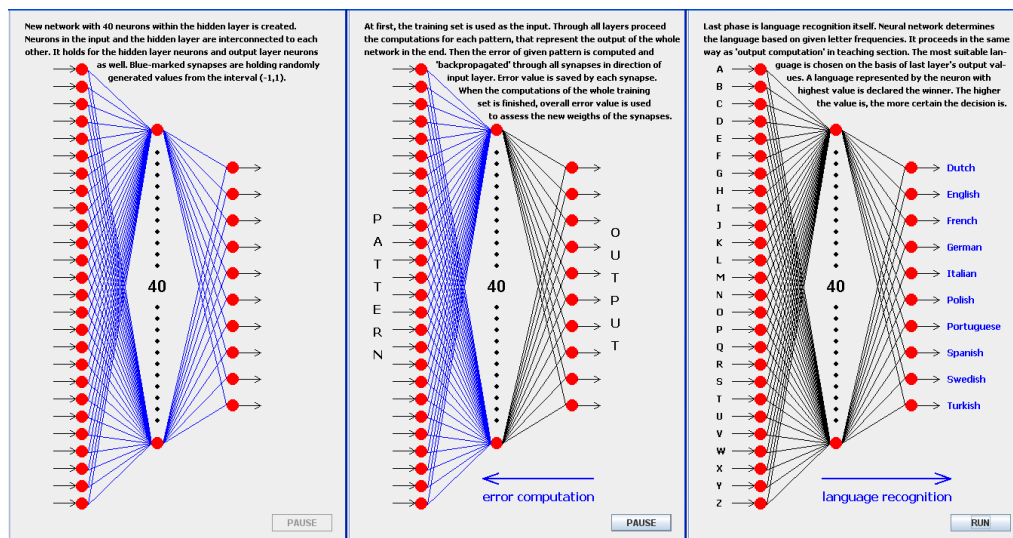


Figure 2.22: "Visualization display" section – "create", "teach" and "recognize" illustrations

At last, let's take a look at the visualization display that forms the whole applet together with the control panel. It doesn't directly interfere with the applet's functionality, because it just serves to visualizing the ongoing partial processes. The figure 2.22 illustrates its phases when the buttons *Create*, *Teach* and *Recognize* are pressed (from left to right respectively). Each of the visualizations includes the explanatory text and an appropriate figure. The "Create" figure is passive, while the "Teach" and the "Recognize" figures act as slow animations. Their pictures change over every four seconds. User can pause and restart the visualization by clicking the button in the lower right corner at any time. The animation is set as running by default.

2.2.4 Experiments with network's setup

When the applet is ready, we can step further to test its functionality by a few experiments. We discuss here how the parameters' settings influence the result. The most correct procedure would be to turn back to the theoretical chapter 2.1.5. and mathematically try to prove the capabilities of the network. Nevertheless, we do not proceed in this way, because it would exceed the scope of the thesis. We are going to focus just on the view from outside and present some hypotheses based on the results of the observations for diverse parameters settings.

Initial configuration

At first, we take a brief look at the first configuration that is randomly generated when the network is created. This is the parameter that user cannot influence, nevertheless it can be very useful to know, what is the highest possible net error evoked by generated weights at the first time.

The first configuration is created by assigning a random value from the interval $[-1, 1]$ to the each weight. As we mentioned, it can be done just by modifying the source code. Thus we set the configuration $\mathbf{w} = (1, 1, \dots, 1)$, because the inputs of all neurons are fully used at this time, therefore the error of the net is the highest possible. It means, that the formula 2.11 dealing with computing the inner potential of the neuron can be modified as follows:

$$\xi = \sum_{i=0}^l 1 \cdot x_i . \quad (2.27)$$

Therefore the inner potential ξ is equal to the sum of its inputs. If we recall the formula computing the output

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}} \quad (2.28)$$

we can see, that the higher the value of ξ is, the less it influences the resulting neuron's output and the closer the output is to the 1. By increasing the number of the hidden neurons, the inner potential grows up and consequently the output of the network converges to 1.

Number of neurons in the hidden layer	Current net error
1	3.4982203849
2	4.0844132089
5	4.4777769609
10	4.4998496886
24	4.4999999999
25	4.5000000000*
50	4.5000000000*
100	4.5000000000*
65535	4.5000000000*

Table 2.3: The net error values produced by the initial configuration $\mathbf{w} = (1, 1, \dots, 1)$.

* This value is rounded up to 4.5 due to variable's capacity. The value just converges to the value 4.5 in fact.

Considering the results presented in the table 2.3 we can add, that the higher the number of hidden neurons is, the less difference in the net error it makes. The accuracy to ten decimal places is not even enough to distinguish the errors of the nets containing 25 hidden neurons and more.

We can say that the potential highest initial net error value is not higher than 4.5 for any number of the neurons in the hidden layer. Hence the error of the new net falls within the interval $[0, 4.5)$.

Number of neurons in the hidden layer

We have to say at the beginning, that it is a hard task to determine the influence of the number of hidden neurons to the network. It is not completely described in the scientific literature till today. Therefore we decided to implement the tests as follows:

1. Generate the network with one neuron in the hidden layer, then with two neurons, etc.
2. For each of the networks check, if the initial error of the net falls within the interval $(0.5, 1.0)$. If not, generate a new net with the same number of hidden neurons. We try to keep similar conditions for all generated networks this way²².
3. Write down the initial net error for the given network and let the network learn in 10 000 cycles with the learning rate equal to 0.01.
4. Note down the current net error. See the graph, whether the value of the net error converges to 0 or higher value. It is highly probable that if the net error does not converge to 0 in 10 000 learning cycles, it will not change later (see the figure 2.23).
5. Continue with the step 1 and add one more neuron into the hidden layer.

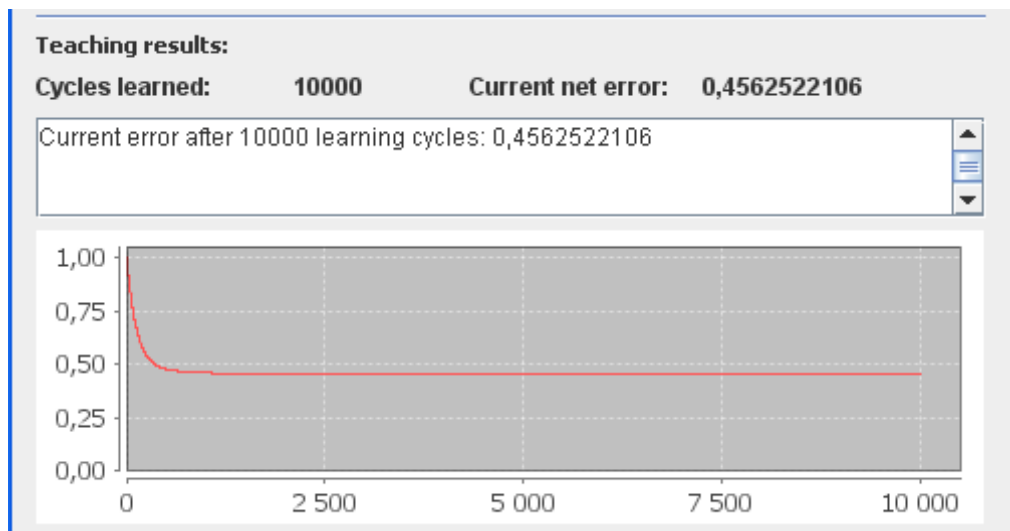


Figure 2.23: Example of a non-convergence to 0 – number of hidden neurons: 2; initial net error: 0.9695094319; learning rate: 0.01

22. We can rewrite the applet's source code to generate a network with an initial configuration such as $\mathbf{w} = (1, 1, \dots, 1)$. Though we acquire the configuration with stable unchanging net error, it does not help us. If all the network weights are the same value, they adapt at the same way (and with same values). It does not lead to convergence to the net error equal to 0, but it stops on higher value.

Number of neurons in the hidden layer	Initial net error	Net error after 10 000 cycles	Convergence to 0
1	0.5272004713	0.4500119399	no
2	0.7419732453	0.4731141539	no
3	0.7329472083	0.7329472083	no
4	0.5057308142	0.4413027152	no
5	0.5911028575	0.0169021708	yes
6	0.7283564301	0.4234247987	no
7	0.8770494018	0.0422623487	yes
8	0.6717495837	0.4446630096	no
9	0.6127882926	0.0299530839	yes
10	0.9335715424	0.0452230382	yes
11	0.7568231373	0.0193755085	yes
15	0.5609105269	0.0269727833	yes
20	0.6439693842	0.0122087790	yes
40	0.5400990343	0.0044145720	yes
100	0.5404234588	0.0011327514	yes
400	0.5125725123	0.0002678197	yes
425	0.5085622729	0.0001190930	yes
450	0.5009852977	0.5000069320	no
500	0.5163263139	0.49999925986	no
1000	0.7439909072	0.5000014892	no
1000	0.5012490964	0.4999996920	no
1000	0.6626659575	0.0000182860	yes

Table 2.4: Results of the test – various number of hidden neurons used

We can infer some hypotheses from the results presented in the table 2.4. There seems to be lower and upper bound of the number of the hidden neurons, that is able to solve the given problem. As we can see, networks with less than 5 hidden neurons do not converge to 0. Following four rows show, that the net converges two times and two times don't. All contiguous networks (with number of hidden neurons equal to or higher than 9) converges to 0. On the other hand, the upper bound seems to be less obvious. The networks with 450 hidden neurons or higher do not tend to converge to 0.

Let's summarize the experiment's results. They can be interpreted just as a help to set up the most optimal number of neurons in the hidden layer, because they heavily depend on the initial configuration (and computed net error therefrom). Small difference of the first configuration causes, that network's error converges to 0 one time and doesn't another (for the same number of hidden neurons). See the last three rows of the table as an example.

It seems, that the language recognition task can be solved by the network containing 10 hidden neurons at least. We think, that this boundary corresponds to the complexity of the task. On the contrary, the upper boundary indicates that the networks with 450 hidden neurons and more do not tend to converge to 0. It is due to too high complexity of the network and inability to adapt the patterns. Noting the results of the experiment, we suppose that sufficient number of the neurons in the hidden layer is 20–100.

Number of learning cycles

An influence of this parameter was broached in the previous section. It holds in general, that the more learning cycles we use, the lower the resulting error of the network is. The correct course of the recognition process is illustrated on the figure 2.24. Nevertheless, the net error does not converge to 0 every time. Considering the figure 2.23 as an example, if the error converges to higher value than 0, the additional learning cycles mostly do not help.

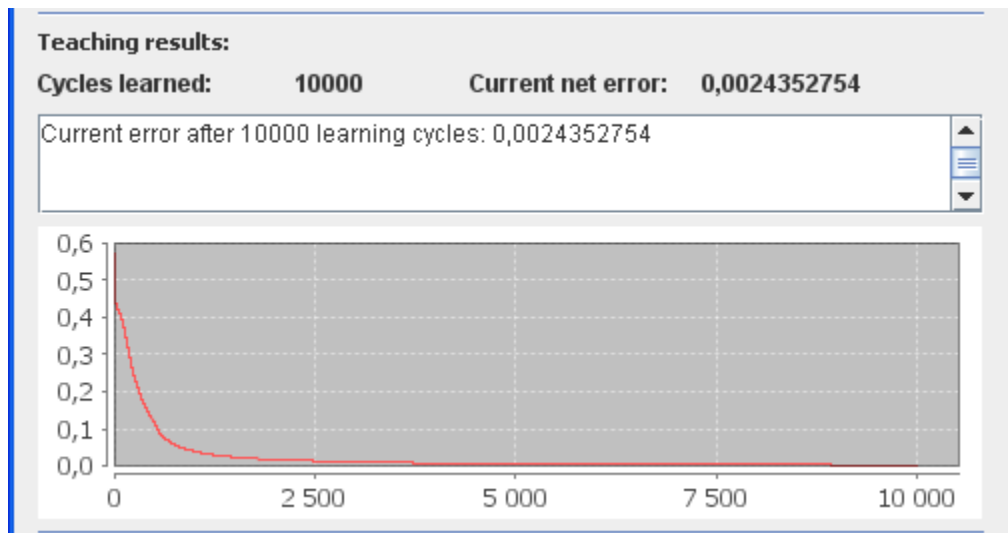


Figure 2.24: Example of a correct course of the recognition process – number of hidden neurons: 40; initial net error: 0.5416811635; learning rate: 0.01

Learning rate

The last parameter, that we discuss here is the learning rate. According to the theory mentioned in the section 2.1.5, the value of the learning rate determines, how influential the alterations of weights at the end of one learning cycle are. Its value has to fall within the interval $(0, 1)$ and it holds, that the higher the value is, the higher the alteration's influence is.

Therefore the high value promises a faster convergence of the net error. Unfortunately, there is a drawback of using high learning rate. A lower value of a net error can be overleapt and start to grow again (see the figure 2.9 on the page 13 and 2.25 on the following page). On the other hand, if too low value is used, the convergence process can be very time-consuming.

Consequently the applicable trade-off has to be found. Therefore we conduct a similar experiment to the recognizing the effect of the various number of hidden neurons presented hereinbefore. Its scope is the observation of an influence of varying learning rate to the speed of net error's convergence. As well as the last time, we set the conditions – the number of neurons in the hidden layer: 40; the initial configuration interval: $(0.5, 1.0)$; the number of learning cycles: 10 000.

The results, presented in the table 2.5, have to be considered tentatively, because they are influenced by varying initial configuration. As we can see, the net error decreases with the increasing learning rate. A value, when the overleaping occurs at the first time, falls within the interval 0.440–0.441. Following values of the learning rate do not converge to 0. Therefore the usable trade-off between speed and the result attainability could be the value 0.2.

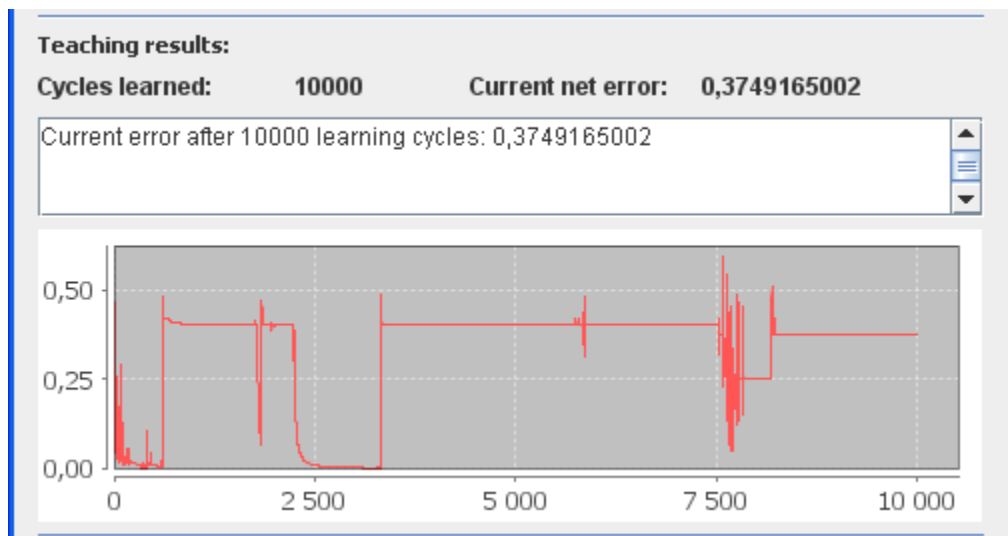


Figure 2.25: Example of a high value of the learning rate – number of hidden neurons: 40; initial net error: 0.5133976759; learning rate: 0.5

Learning rate	Initial net error	Net error after 10 000 cycles
0.001	0.5963302914	0.0356159682
0.01	0.5519294621	0.0040699484
0.05	0.9084064743	0.0008747488
0.1	0.6968796094	0.0003196628
0.3	0.6010278287	0.0001462317
0.44	0.9653188698	0.0001407894
0.441	0.5998601806	0.4999664417
0.445	0.5147035657	0.3750413861
0.45	0.5551048284	0.4001229088
0.5	0.5748253598	0.3734122487
0.7	0.6668643170	0.4999999195
0.9	0.7598394836	0.4500009156

Table 2.5: Results of the test – various learning rate used

Recognition process

At last, we just briefly summarize the potential of the applet. We decided not to present an embrasive evaluation of the success of the recognition, because it highly depends on two main parameters – a network, that is used (mainly on the value of the net error) and a text, that is intended to being processed. Therefore, we just bring here the results of the experiment with a random initial configuration and a default settings. After learning phase, we let the network recognize ten files containing a short fragment of the text gathered from national versions of the Wikipedia²³. The shortest one has got 881 characters, while the longest

23. These texts were displayed on the main site of the Wikipedia's national version websites on 14th November 2009.

one contains 1580. It should be enough for the applet to work properly. The files are a part of the enclosed compact disk and they can be downloaded on the applet's internet page as well. Hence, the settings of the applet was following:

- *Number of neurons in the hidden layer* = 40
- *Initial net error* = 3.9242268992 (randomly generated configuration of the network)
- *Learning rate* = 0.01
- *Net error after 10 000 learning cycles* = 0.0048374695

Text file	Number of characters	Result	Recognized correctly
test_dutch.txt	1580	0.4003799379	yes
test_english.txt	1092	0.8306231621	yes
test_french.txt	924	0.3237752449	yes
test_german.txt	1069	0.8066869342	yes
test_italian.txt	1310	0.6125173902	yes
test_polish.txt	1030	0.8086577918	yes
test_portuguese.txt	1330	0.4423335938	yes
test_spanish.txt	881	0.6204429594	yes
test_swedish.txt	931	0.5648895081	yes
test_turkish.txt	1816	0.8280184014	yes

Table 2.6: Results of the recognition process

All the recognition experiments presented in the table 2.6 were successful. The worst result was performed by the french text (0.3237752449) and the best was the english one (0.8306231621). Next testing is left on the user.

2.2.5 User's guide

The last section serves as a short user's guide for those, that just want to start playing with the applet without knowing the details presented in the previous sections of the chapter 2.2.

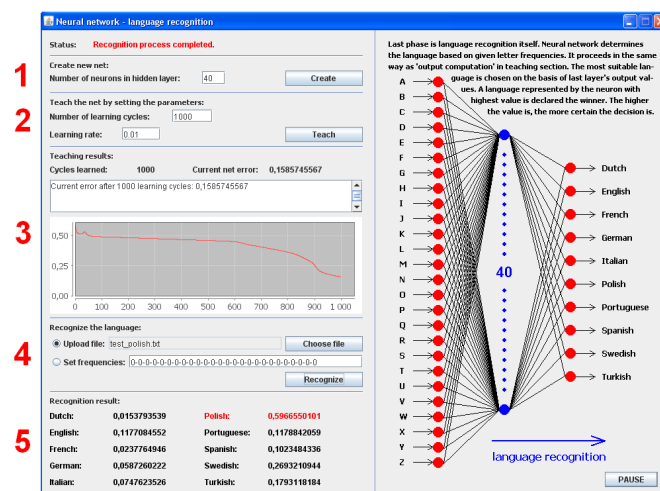


Figure 2.26: User's guide: Create – Teach – Teaching results – Recognize – Recognition results

Please, watch the *Status* in the top left part of the applet, it informs about current activity. The right half of the applet serves for a visualization of its functionality. It starts and subsequently changes automatically, when appropriate button is clicked. You can pause it and rerun at any time.

1. Set a *number of neurons in the hidden layer*. The default value is 40. A recommended range is 20–60. Click the button *Create* to establish a new network afterwards.
2. Set the parameters of the teaching process – a *number of learning cycles* and a *learning rate*. The higher the chosen learning rate is, the longer the teaching process takes. It is better to start with lower values and repeat it more times. Default value is 10 000 and we do not recommend to use more than 50 000 cycles at once. Learning rate is preset on the value 0.01. Do not use higher values than 0.3 – it tends not to find the correct result. At last, click the button *Teach* to start the teaching process.
3. When the network is taught, see its result. A *current net error* indicates the quality of the network. The lower the value is, the higher chance for correct recognition of the language is. A *graph* that figures a course of the teaching process is worthwhile to see as well. You can continue in teaching the network or take a step forward to the recognition process.
4. There are two options how to upload the data into the applet. The easier one is to click the button *Choose file* and find the requested text file. The little more difficult is to enter the vector of 26 numbers, that represents the frequency of the letters occurrence. See the section 2.2.3 for more details about the manual entry. Click the *Recognize* button afterwards.
5. See the results of the recognition process. The higher the value is, the more reliable is the result. The highest value is highlighted in a red color and declared the winner.

Feel free to play with the network's parameters and watch how they influence the recognition process. Check the section 2.2.4, where a few experiments and their results are discussed.

Chapter 3

Genetic algorithms

The thesis deals with a topic of genetic algorithms in its second part. As well as the first one, this part is divided into two sections. At first, it introduces a biological inspiration and a history of the genetic algorithms. A theoretical background is clarified afterwards.

The second section describes an application of the genetic algorithms approach on an optimization task called *Robby, the soda-can-collecting robot*. The solution is discussed in detail and finally presented in the form of a Java applet.

3.1 Theory

3.1.1 Biological inspiration

Genetics is a discipline of biology focusing on a research of heredity and a variation in living organisms. The name is derived from a unit of heredity called *gene* and was established by William Bateson¹ in the year 1906. Genetics is rapidly developing science nowadays.

Let's start with an *evolution theory* which forms a so-called wrapper to the genetics. It treats of birth and evolution of a life. Many theories came up during the past, but nowadays the term of evolution theory denotes a research based on the work of Charles Darwin² and Alfred Russel Wallace³. They brought in the idea, that the evolution proceeds by small alterations in organisms during generations. These alterations are set so that good traits of ancestors are retained by their offspring. This concept is called *Darwinism*.

Darwin and Wallace drew up their work without having a deeper knowledge of molecular biology. Independently of them, a czech naturalist Gregor Johann Mendel⁴ founded a new scientific discipline – *genetics*. Genetics deals with a heredity on a level of cells, hence it just confirmed the ideas of Darwinism and provided a new knowledge for deeper research. A synthesis of darwinism and genetics is collectively called *Neo-Darwinism*.

A *cell* is a functional basic unit of all known living organisms (see the figure 3.1). Its the most important component from the genetic point of view is a *nucleus*. The nucleus contains *chromosomes*, that are organized structures of *Deoxyribonucleic acid (DNA)* and proteins. DNA consists of ordered sequence of *nucleotides*⁵ that holds the hereditary information.

1. William Bateson (1861 – 1926) was a british geneticist. He defined genetics as "Study of plant breeding" originally and the word was used for the heredity of all organisms later.

2. Charles Darwin (1809 – 1882) was a British naturalist. He is considered as a founder of an evolution biology.

3. Alfred Russel Wallace (1823 – 1913) was a British naturalist and biologist that originally worked on a theory of evolution independently of Charles Darwin, but later became his cooperator.

4. Gregor Johann Mendel (1822 – 1884) was an Augustinian priest. He is known as "father of modern genetics" nowadays. Mendel became famous posthumously as late as William Bateson translated his works into English and popularised them.

5. Nucleotides consists of one of five nitrogenous bases – cytosine (C), guanine (G), adenine (A), thymine (T) and uracil (U). It identifies a type of nucleotide's information.

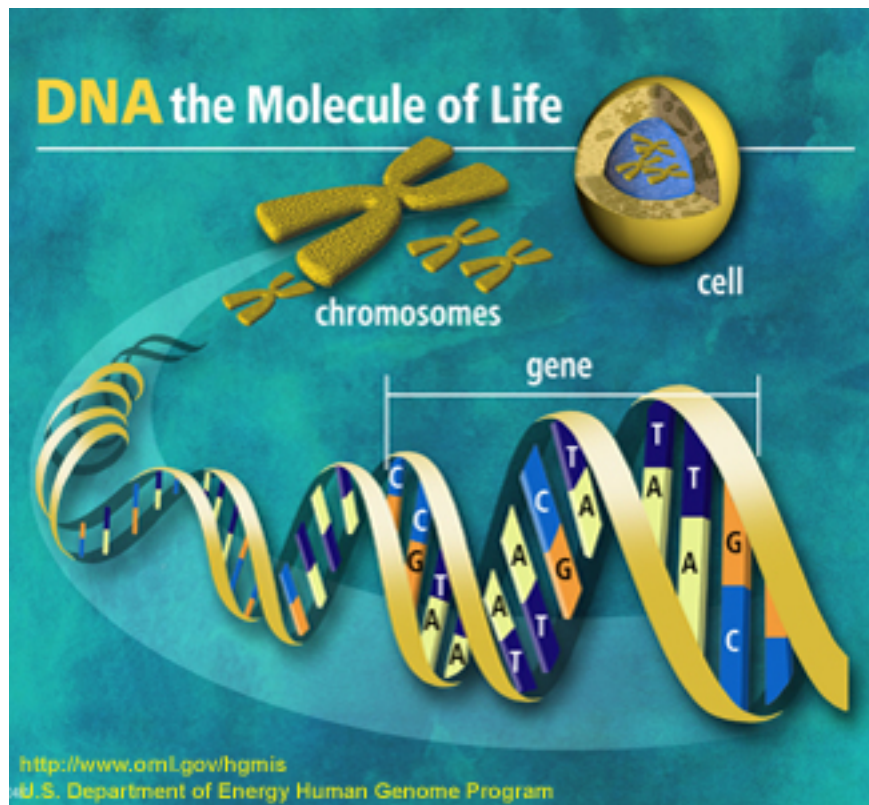


Figure 3.1: A diagram of the cell's components important from the genetic point of view
 source: U.S. Department of Energy Genome Programs, <http://genomics.energy.gov>

The description of a cell is by far not exhaustive, but it should be enough for our purposes.

At last, we have to mention a reproduction process of cells. It is a complicated procedure, therefore we just notify here parts that are interesting for us from a computational perspective. The reproduction of gametes is called *meiosis*⁶ and consists of four phases⁷. Simply speaking, chromosomes are separated and form new cells then. During the meiosis, an operation called *crossover* takes place – two chromosomes mutually swap parts of their DNA. Rarely happens another genetic operation – a *mutation*. The mutation stands for a change of types of nucleotides in DNA. These two operations were mentioned, because they served as a pattern for computational genetic algorithms.

3.1.2 History

First mention of genetic algorithms' ancestors can be found around the year 1960, when evolutionary biologists were inquiring a potential of involving computers into their research. Hence it was necessary to formalize a natural evolution. At that time, researchers did not have any idea, that they just laid foundations of a computational technique applicable to solving a broad set of problems.

Nevertheless, Ingo Rechenberg⁸, a german scientist, introduced a technique named by

6. Other body cells are reproduced by a process called *mitosis*.

7. The phases in turn are: *prophase, metaphase, anaphase, telophase*.

8. Ingo Rechenberg (1934) is a german computer scientist and professor at the Technical University of Berlin.

him as the *evolution strategy*. This procedure was still quite far from that, what we call genetic algorithm today, but the concept was promising. He generated one random initial individual and with use of the operation of mutation evolved one offspring, compared them to one another and used the better of them for next rounds of evolution. Hence, a usage of population instead of one individual and the crossover operation was not known yet.

In the year 1966 Lawrence J. Fogel⁹ presented a book *Artificial intelligence through simulated evolution*¹⁰[11] where he introduced a technique called *evolutionary programming*. It derived benefit from usage of finite-state automata and Rechenberg's evolution strategy. Thus, the terms of population and crossover stayed still undiscovered.

That changed John H. Holland¹¹ by introducing his book *Adaptation in natural and artificial systems*[12] in the year 1975, where he presented a usage of populations, crossover and other recombination operators. Thereby he described genetic algorithms how they are known today.

Since then an interest in this technique rapidly increased. By the early 1980s, genetic algorithms were used in a broad range of tasks – from mathematical NP-problems like bin-packing or graph coloring to engineering issues such as pipeline flow control. A few years later a genetic algorithms fever started to spread behind borders of university laboratories into the commercial sector. It caused, that evolutionary computing found its place in various, earlier unimaginable sectors – biochemistry, molecular biology, aerospace engineering, microchips design, stock market predictions, scheduling at airports and many others.

3.1.3 General concept of genetic algorithms

We fragmentary alluded to genetic algorithms in term of computer science in the last two sections. Hence, it will be very useful to sum up a general concept and provide a comprehensive sight of them.

With regard to natural evolution, we decompose the process into three compact parts (see the figure 3.2 on the page 38):

- *Initialization* consists of setting paramters (e.g. size of population) and determining encoding sample. Another important part is to shape a fitness function. Finally, an initial generation can be created.
- *Selection* is a subprocess, when the individuals in a current generation are evaluated and ranked. Consequently a probability of selection is assigned to each of them. It proceeds once for a generation.
- *Reproduction* is a part responsible for picking out a pair of individuals in accordance with the probabilities of selection at first. Afterwards, the chosen individuals are reproduced by applying genetic operations of crossover and mutation on them. Resulting offsprings become members of a next generation. A reproduction proceeds as long as the next generation does not have a same size as the current.

Following sections go into details of the aforementioned parts of a genetic algorithm and clarify all their aspects.

9. Lawrence J. Fogel (1928 – 2007) was a pioneer in evolutionary computation.

10. It was co-authored by Alvin J. Owens and Michael J. Walsh.

11. John Henry Holland (1929) is an American scientist and Professor of psychology, electrical engineering and computer science at the University of Michigan. He is known as the father of genetic algorithms.

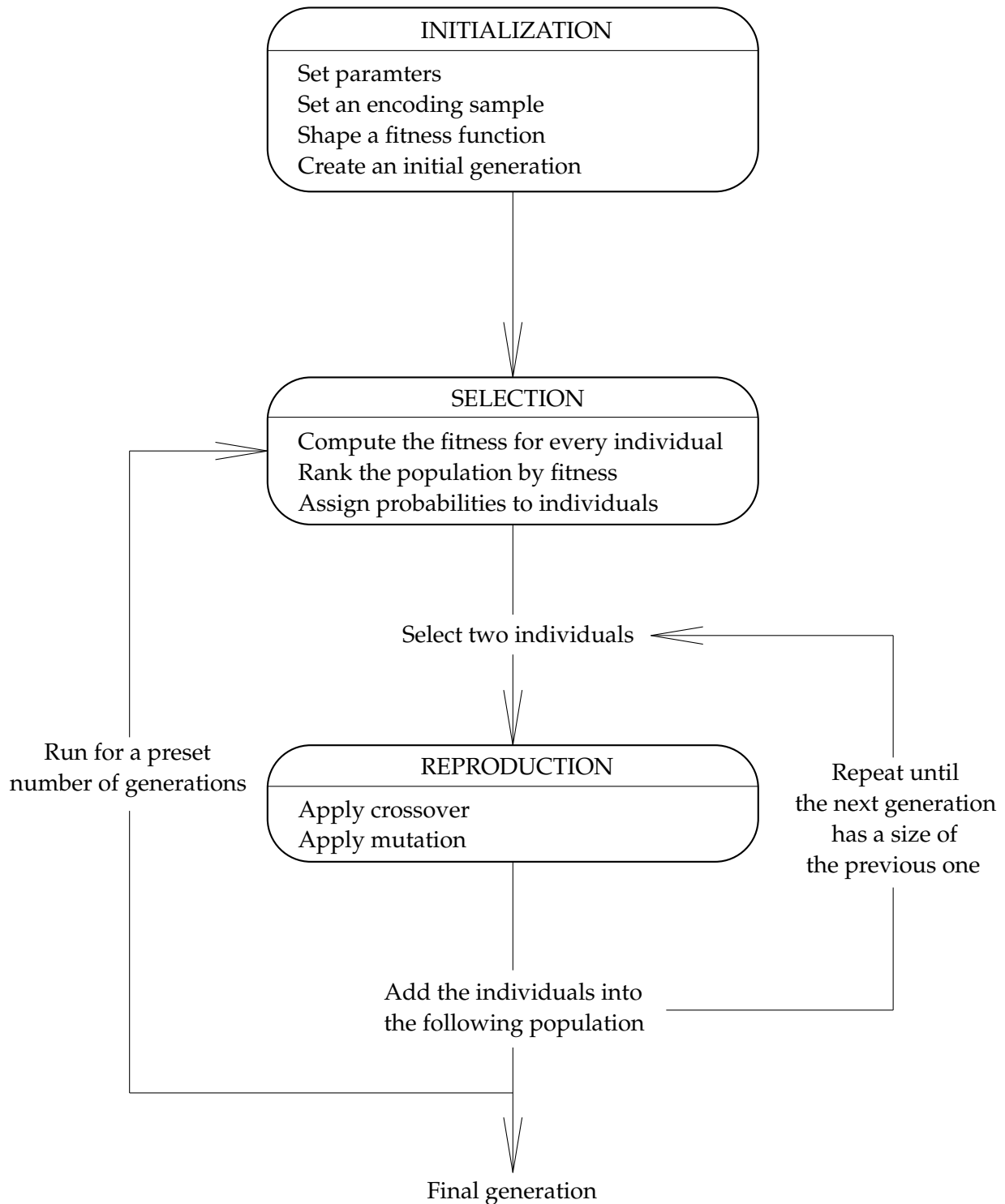


Figure 3.2: General schema of a genetic algorithm

3.1.4 Initialization

A first phase of a genetic algorithm is called initialization. It proceeds just once for the whole algorithm and right on its beginning.

Parameters

A few parameters have to be set at first – *population size*, *number of generations* and *mutation probability*.

- *Population size* stand for a number of individuals in a population. A population contains same number of individuals in every generation.
- *Number of generations* determines how many times a new set of individuals should be reproduced from the current one.
- *Mutation probability* specifies a chance, that an operation of mutation will be used during the reproduction phase. It will be discussed in a greater depth in a description of that phase.

Encoding sample

Each individual is identified by its DNA in a nature. Being able to start a computational genetic algorithm, each individual of a problem's search space has to be encoded into a string of integers¹². Let's illustrate a simple encoding example:

We want to use a genetic algorithm for evolving the word *baggage* from scratch. A search space of the problem consists of seven characters long strings, where each of their letters falls into an english alphabet a, b, \dots, z . Considering that the highest letter from the beginning of the alphabet in the word *baggage* is g , we can reduce each character's range to a, b, \dots, g . It does not mean big savings on a time of a computation in this case, however it can be significant in real problems.

Encoding sample puts up itself: $a = 1, b = 2, c = 3, d = 4, e = 5, f = 6, g = 7$. One can argue, that there is no need to use any encoding into integers. Yes, it is true for our simple example. Nevertheless, a usage of an integer encoding is very helpful during the implementation and crucial when solving a real task.

Hence, the search space of our problem is a set of 7^7 strings from 1111111 to 7777777, where the word *baggage* is represented as 2177175.

Fitness function

Continuing with our example, we need a formula for ranking the individuals within a generation. It is called *fitness function*. The higher a functional value of the fitness function is, the higher the current individual is ranked. Therefore, we shape the function as follows:

$$f(\text{individual}) = \frac{\text{number of correctly located integers in an encoded individual}}{\text{length of an encoded individual}}. \quad (3.1)$$

The more correctly located integers (encoding letters) in the string are, the closer the string's fitness value to 1 is. The fitness value for a word *baggage* equals to 1.

12. It is possible to use another type of representation – e.g. floating-point numbers. However, it brings difficulties and fits better into a concept of the evolutionary programming.

Random initial generation

Before moving to the *selection* phase, an initial generation of a population is created. Each of *population size* individuals is chosen at random.

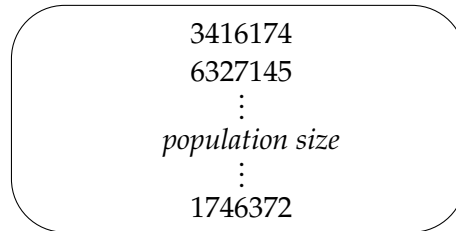


Figure 3.3: Initial generation of a population in the *baggage example*

3.1.5 Selection

After the *initialization* phase follows the *selection* phase. It proceeds once for a generation, thus *number of generations* times for a whole algorithm. It can be perceived as an initialization of a generation, because the individuals are being prepared for the *reproduction* phase here.

Compute the fitness

At first, the fitness value is computed for every individual in the current generation.

Rank the population by fitness

Next step is to sort the individuals by fitness. The main idea of the fitness function is to find out, which individuals are the best in the current generation and give them higher chance to be reproduced and included in the next generation.

Assign probabilities to individuals

A last task of the *selection* phase is closely associated with ranking the population. There are several methods for the individuals' selection. One of them is a *roulette wheel method*.

The roulette wheel method stands for an approach, where a probability of selection based on the fitness value is assigned to each individual. The higher the fitness value is, the higher probability of being selected the individual has. The inspiration by a roulette wheel in a casino is quite clear. A sum of fitness values of all individuals in a generation is considered as the whole wheel. A proportion of the wheel is assigned to each of the individuals based on their fitness values.

Let's consider a generation of the size 3 in our *baggage example*. Its members are: **3416174**, **6327145** and **1746372**. The bold digits are correctly located in the requested word *baggage* (encoded as 2177175). Therefore the fitness values of the strings are 0.2857, 0.4286 and 0.1429 respectively (rounded to four decimals). When normalizing them to percents, we get 0.3333, 0.5000, 0.1667 as chances, that a given individual will be chosen (see the overview of the computation in the figure 3.4 on the page 41).

At last, we just mention a few other selection methods¹³. A *tournament selection method* is often used in practice. Individuals take part in several tournaments and then they are ranked based on the results. Tournaments can have a various form. Among simpler methods are *stochastic random* or *top percent* (just a top percent of individuals has a chance to be selected).

13. Consider that all the methods are not applicable to all tasks.

Individuals	Fitness	Probability	Roulette wheel
3416174 ...	$\frac{2}{7} = 0.2857 \dots$	$\frac{0.2857}{0.8572} = 0.3333$	
6327145 ...	$\frac{3}{7} = 0.4286 \dots$	$\frac{0.4286}{0.8572} = 0.5000$	
1746372 ...	$\frac{1}{7} = 0.1429 \dots$	$\frac{0.1429}{0.8572} = 0.1667$	
$\Sigma = 0.8572$			

Figure 3.4: Overview of the probabilities computation in the *baggage example*

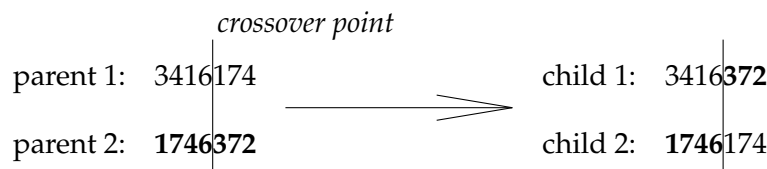
3.1.6 Reproduction

The last phase is called *reproduction*. It begins with a random selection of two individuals based on probabilities computed in the *selection* phase¹⁴. The chosen pair will be modified using two genetic operators – *crossover* and *mutation*. The two descendants created this way become members of a next generation.

The *reproduction* phase is repeated until a next generation contains the same number of individuals as the current one.

Crossover

The first genetic operator is called *crossover*. It is applied on the both selected individuals at once. When their strings are crossed, the result is two new strings, where each of them contains a substring of each of the parents. Let's use our *baggage example* for make the operator's working clear.

Figure 3.5: Crossover operator forming offsprings in the *baggage example*

At first, a *crossover point* is randomly chosen. The point represents such position in individuals' strings, that the subchains before and behind the position are concatenated in a way, that each descendant's string consists of substrings inherited from both parents.

Mutation

The *crossover* is followed by another genetic operator – *mutation*. In contrast to the *crossover*, the *mutation* is applied just on a single individual. In our case, it proceeds on the selected pair of individuals in turn without influencing each other.

Among the parameters defined in the *initialization* phase is a *mutation probability*. Each single position of the individual's string has got the *mutation probability*, that a new random value is generated and assigned to the current position (see the figure 3.6 on the page 42).

14. One can argue that this action fits better into the previous phase by reason of its name, however it directly precedes the *reproduction* phase and functionally stands between them. Therefore, we decided to place it into this phase.

1746174 \longrightarrow 1146574

New random values assigned to positions 2 and 5

Figure 3.6: Mutation operator forming an offspring in the *baggage example*

3.1.7 Summary

We presented a general concept of genetic algorithms in the subsections 3.1.3 – 3.1.6 (see the summary in a box below). It should be noted, that it naturally does not provide exhaustive description of the topic. There are a lot of modifications based on the *initialization–selection–reproduction* backbone. We just demonstrated the aforementioned theory with regard to an implementation part of the thesis presented in the section 3.2.

For those that are interested in additional information and deeper studies in the field of genetic algorithms we recommend a few titles. *The computational beauty of nature*[14] written by Gary William Flake introduces computer methods and their inspiration in the context of nature. On the other hand, Rolf Pfeifer’s and Josh Bongard’s book *How the body shapes the way we think*[15] deals, among others, with evolutionary methods used in robotics and artificial intelligence. Those titles just illustrate how widely genetic algorithms can be used.

1. Set paramters *population size, number of generations, mutation probability*.
2. Set an encoding sample.
3. Shape a fitness function.
4. Create an initial generation.
5. For *number of generations* do
 - (a) Compute the fitness function for every individual.
 - (b) Rank the population by fitness.
 - (c) Assign probabilities to individuals.
 - (d) For *population size/2** do
 - i. Randomly select two individuals based on computed probabilities.
 - ii. Apply crossover on them.
 - iii. Apply mutation in turn on both of them, use *mutation probability*.
 - iv. Put the offsprings into a new generation.
6. Get the final generation.

* If *population size* is odd, a small modification has to be made in the last loop of the *reproduction* phase, when there is only one place left in a new population. It proceeds as usual during the steps i–iii. When putting the offsprings into a new generation in the step iv, just a first descendant is used. The second is discarded.

3.2 Application – Robby, the soda-can-collection robot

Section 3.2 is a demonstration of a genetic algorithm's usage in practice. The section presents a solution of a task firstly published in a course CS 441/541 Artificial intelligence taught by Professor Melanie Mitchell at the Portland State University. The task was chosen for its clearness and attractive assignment, that directly suggests itself to introduce genetic algorithms in an interesting form.

We are going to follow the schema used in the section 2.2, where a language recognition applet is presented. At first, we define an assignment of the task. Then follows a concept of a solution. After that, a result in a form of Java Applet is introduced. Next to the last section is devoted to a demonstration of applet's capabilities and finally, the last part contains a brief user's guide.

3.2.1 Assignment

As we mentioned, the task we chose to implement was published as a homework in the course CS 441/541 Artificial Intelligence[16] taught by Melanie Mitchell¹⁵. Therefore, we present the assignment in an unchanged form[17]:

Robby, the Soda-Can-Collecting Robot

Robby's job is to clean up his world by collecting the empty soda cans. Robby's world, illustrated in figure 3.7, consists of 100 squares (sites) laid out in a 10×10 grid. You can see Robby in site 0,0. Imagine that there is a wall around the boundary of the entire grid. Various sites have been littered with soda cans (but with no more than one can per site).

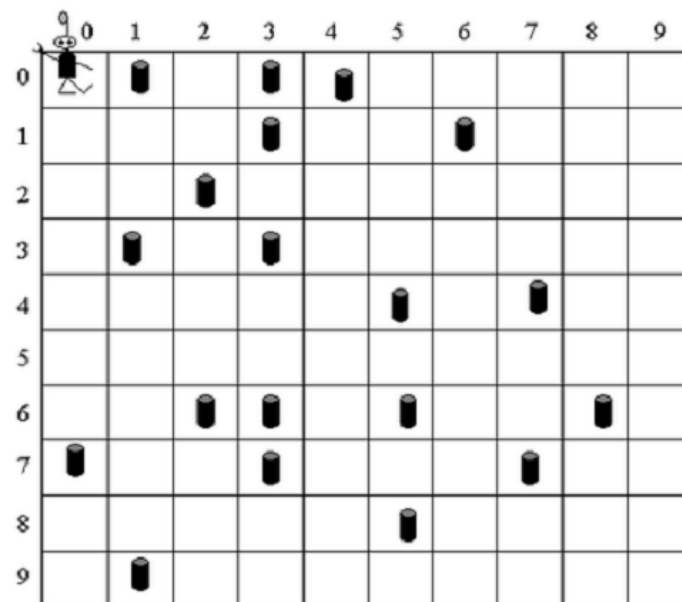


Figure 3.7: Robby's world: A 10×10 array strewn with empty soda cans on random positions
source: CS 441/541: Artificial Intelligence – Homework 6[17]

15. Melanie Mitchell is a professor of computer science at Portland State University. Her major work areas are complex reasoning, complex systems, genetic algorithms and cellular automata.

Robby isn't too intelligent, and his eyesight isn't that great. From wherever he currently is, he can see the contents of one adjacent site in the north, south, east and west directions, as well as the contents of the site he's currently in. A site can be empty, contain a can, or be a wall. For example, in the figure 3.7, Robby, at site 0,0, sees that his current site is empty (i.e., contains no soda cans), the "sites" to the north and west are walls, the site to the south is empty, and the site to the east contains a can.

For each cleaning session, Robby can perform exactly 200 actions. Each action consists of one of the following seven choices: move to the north, move to the south, move to the east, move to the west, choose a random direction to move in, stay put, or bend down to pick up a can. Each action may generate a reward or a punishment. If Robby is in the same site as a can and picks it up, he gets a reward of 10 points. However, if he bends down to pick up a can in a site where there is no can, he is fined 1 point. If he crashes into a wall, he is fined 5 points and bounces back into the current site.

Clearly, Robby's reward is maximized when he picks up as many cans as possible, without crashing into any walls or bending down to pick up a can when no can is there.

The task is to write code for a genetic algorithm to evolve control strategies for Robby the Robot.

3.2.2 Solution concept

The task assignment[17] includes also a concept of a solution. Hence, we took that concept over and complemented it with selected parts of a book *Complexity: A guided tour*[18] written by Melanie Mitchell. The autor elaborates the solution in greater depth there. This section contains an extraction from these two sources (italics) completed with our comments and specifications (base font).

Robby's strategy should be encoded as a look-up table that gives, for every possible state – i.e., situation he can encounter – the action he should take when in that state.

There are five different sites (north, south, east, west, current), each with three possible types of contents (wall, empty, can). Thus there are $3 \times 3 \times 3 \times 3 \times 3 = 243$ different "possible" situations. Of course this number includes some "impossible" situations, such as those in which Robby's current site contains a wall (since Robby will always bounce back from a wall). However, these don't have to be filtered out; they can be just left in the table, since they won't have much affect on the GA.

Here's an example of a strategy – actually, only part of a strategy, since an entire strategy would be too long to list here.

<i>Situation</i>					<i>Action</i>
<i>North</i>	<i>South</i>	<i>East</i>	<i>West</i>	<i>Current site</i>	
<i>Empty</i>	<i>Empty</i>	<i>Empty</i>	<i>Empty</i>	<i>Empty</i>	<i>Move north</i>
<i>Empty</i>	<i>Empty</i>	<i>Empty</i>	<i>Empty</i>	<i>Can</i>	<i>Move east</i>
<i>Empty</i>	<i>Empty</i>	<i>Empty</i>	<i>Empty</i>	<i>Wall</i>	<i>Move random</i>
<i>Empty</i>	<i>Empty</i>	<i>Empty</i>	<i>Can</i>	<i>Empty</i>	<i>Pick up can</i>
<i>⋮</i>	<i>⋮</i>	<i>⋮</i>	<i>⋮</i>	<i>⋮</i>	<i>⋮</i>
<i>Wall</i>	<i>Empty</i>	<i>Can</i>	<i>Wall</i>	<i>Empty</i>	<i>Move west</i>
<i>⋮</i>	<i>⋮</i>	<i>⋮</i>	<i>⋮</i>	<i>⋮</i>	<i>⋮</i>
<i>Wall</i>	<i>Wall</i>	<i>Wall</i>	<i>Wall</i>	<i>Wall</i>	<i>Stay put</i>

Table 3.1: Example of a strategy, source: CS 441/541: Artificial Intelligence – Homework 6[17]

For example, Robby's situation in figure 3.7 is

North	South	East	West	Current site
Wall	Empty	Can	Wall	Empty

A situation is a pentad of four sites around Robby completed with his current site. Handling the pentad in its verbal representation seems to be unpractical, therefore we turn it into a numerical form (see the table 3.2).

Site	Numerical representation
Empty	0
Can	1
Wall	2

Table 3.2: Numerical representation of sites

Hence, Robby's situation in the figure 3.7 is numerically represented as a string 20120. Thereby we obtain a format which is easy to represent in any data structure when programming.

Robby's world can be converted into a numerical form as well. We extend the original 10×10 array to 12×12 grid. Two added rows and columns stand for a border of Robby's world (see the figure 3.8 representing the figure 3.7 in numerical format).

2	2	2	2	2	2	2	2	2	2	2	2
2	0	1	0	1	1	0	0	0	0	0	2
2	0	0	0	1	0	0	1	0	0	0	2
2	0	0	1	0	0	0	0	0	0	0	2
2	0	1	0	1	0	0	0	0	0	0	2
2	0	0	0	0	0	1	0	1	0	0	2
2	0	0	0	0	0	0	0	0	0	0	2
2	0	0	1	1	0	1	0	0	1	0	2
2	1	0	0	1	0	0	0	1	0	0	2
2	0	0	0	0	0	1	0	0	0	0	2
2	0	1	0	0	0	0	0	0	0	0	2
2	2	2	2	2	2	2	2	2	2	2	2

Figure 3.8: Numerical representation of figure 3.7, Robby's position is bold

Robby never enters site evaluated by 2. It just serves as an indicator, when Robby tries to move into a wall. When it happens, Robby stays put and is fined 5 points.

Therefore it is easy to keep the current state of Robby's world this way. When Robby performs an action, we just simply alter affected sites.

To decide what to do next, Robby simply looks up this situation in his strategy table, and finds the corresponding action to take.

The "chromosome" to be evolved by the GA is just a listing of the 243 actions in the rightmost column of the strategy table, in the order given. The actions are numbered as follows:

<i>Action</i>	<i>Numerical representation</i>
<i>Move north</i>	0
<i>Move south</i>	1
<i>Move east</i>	2
<i>Move west</i>	3
<i>Move in a random direction</i>	4
<i>Stay put</i>	5
<i>Pick up can</i>	6

Table 3.3: Numerical representation of actions

Then the chromosome representing the strategy demonstrated in the table 3.1 would be

0246 ... 3 ... 5

The GA remembers that the first action in the string (here action 0: Move north) goes with the first situation (“Empty Empty Empty Empty Empty”), the second action (here action 2: Move east) goes with the second situation (“Empty Empty Empty Empty Can”), and so on. In other words, the situations corresponding to these actions don’t have to be explicitly listed; instead the GA remembers the order in which they are listed.

We decided not to keep exactly the order outlined in the table 3.1. Corresponding numerical formats of situations stand in following order in our implementation: 00000, 10000, 01000, ..., 00001, 20000, ..., 00002, ..., 00002, 11000, 10100, ... 22222. The more number of zeros in the situation are, the earlier in the chromosome it occurs.

Anyway, different order of situations does not change anything in following lines that specify the genetic algorithm solving Robby’s problem. It fully conforms to the theory described in the sections 3.1.3 – 3.1.7.

Generate the initial population. The genetic algorithm starts with an initial population of POPULATION_SIZE random individuals (strategies). As described above, each individual strategy is a list of 243 numbers, each gene between 0 and 6, which stands for an action (0 = Move north, 1 = Move south, 2 = Move east, 3 = Move west, 4 = Stay put, 5 = Pick up can, and 6 = Random move). In the initial population, these numbers are filled in at random.

Repeat the following for NUM_GENERATIONS generations:

1. Calculate the fitness of each individual in the population. The fitness of a strategy is determined by how well the strategy lets Robby do on NUM_SESSIONS different cleaning sessions. A cleaning session consists of putting Robby at site 0, 0, and throwing down a bunch of cans at random (each site can contain at most one can; the probability of a given site containing a can is 50%). Robby then follows the strategy for NUM_ACTIONS_PER_SESSION actions in each session. The score of the strategy in each session is the number of reward points Robby accumulates minus the total fines he incurs. The strategy’s fitness is its average score over the NUM_SESSIONS different cleaning sessions (each of which has a different configuration of cans).
2. **Rank the population by fitness.** Assign to each individual in the population an integer: 1 for the individual with highest fitness, 2 for the individual with next highest fitness, and so on. Any ties can be broken at random.

3. **Apply evolution** to the current population of strategies to create a new population. That is, repeat the following until the new population has `POPULATION_SIZE` individuals:
 - Choose two parent individuals from the current population probabilistically based on fitness rank. In more detail, the probability that each individual will be chosen is equal to:

$$\frac{POPULATION_SIZE - fitness_rank + 1}{1 + 2 + \dots + POPULATION_SIZE}$$
 - Mate the two parents to create two children. That is, randomly choose a position in each number string; form one child by taking the numbers before that position from parent A and after that position from parent B, and vice versa to form the second child.
 - With a probability `MUTATION_PROBABILITY`, mutate numbers in each child. That is, for each number in child's chromosome, with probability `MUTATION_PROBABILITY` replace that number with a randomly generated number between 0 and 6.
 - Put the two children in the new population.
4. Once the new population has `POPULATION_SIZE` individuals, return to step 1 with this new generation.

According to the step 1, `NUM_SESSIONS` different *playgrounds*¹⁶ are generated during a computation of the fitness function. There is not explicitly mentioned in the assignment, if the playgrounds in cleaning sessions have to be exactly the same for all strategies in a generation or just with same properties (especially a probability of a can occurrence). We decided to choose the second option. Hence, each strategy in a generation is tested in a set of `NUM_SESSIONS` randomly generated playgrounds. We think, that it helps to find globally best strategies and not only strategies, that are successful in the same given set of playgrounds.

Following section presents a resulting applet, where the concept is specified and explained deeper thereon as necessary.

3.2.3 Resulting applet

Resulting applet section proceeds in a same manner as the section 2.2.3 devoted to a presentation of the language recognition applet. A whole applet is described at first. Subsequently, we continue with sections where each of them deals with an applet's part solving particular subtask.

Before we start with a description, we would like to advert to a section 3.2.5 named *User's guide*, where a brief applet's manual is presented for those that are not interested in the applet's functionality in a greater depth.

Applet as a whole

The whole applet consists of three optically separated parts – *control panel*, *playground visualization* and *theory visualization* (see the picture 3.9 on the page 48).

Control panel on the left half of the applet is used to regulate its whole functionality by setting parameters. Playground visualization located on the right half of the applet illustrates

16. We use a term *playground* for a grid randomly strewn with cans corresponding to Robby's world.

Genetic algorithm - Robby, the soda-can-collecting robot

Status: New playground was successfully created.

Create a world:
 Population size: Number of sessions: Create a world
 Number of actions per session:

Evolution settings:
 Number of generations: Mutation probability: Evolve

Generation 997: 187.8 184.8 182.48 182.4 182.3 182.1 181.8 181.7 181.5 181.26 180.9 180.87
 Generation 998: 184.8 184.0 183.34 183.3 183.1 182.7 182.2 182.0 182.0 182.0 181.7 181.5 1
 Generation 999: 184.4 183.8 183.8 181.9 181.7 181.5 181.4 180.3 180.28 180.24 180.1 179.9
 Generation 1000: 187.78 186.6 186.2 186.0 181.96 181.9 181.8 181.6 181.5 181.38 181.2 181

Best evolved strategy: 193.9

Strategy chain:
 401236103131236236366321234006032162265660324310243106266366310545316035505603666
 116366466305024626512151623015426505464616621563512603321523266063220541236521264
 141113633201135366653360445650003043023102422546333331600111262045024404331224302

Create a playground

Test strategy

Stop testing procedure

Strategy chain:

Step counter: 0 **Action:** none
Total score achieved: 0

A crossover is a first genetic operation applied on the chosen strategies. A position in the strategies' chains is randomly chosen and the subchains before and behind the position are concatenated in a way, that they form new chains.

1232425063 ... 43233253412123240342 ... 3321325341

2610124554 ... 63432034053245021420 ... 3560110324

↑

1232425063 ... 43233253413245021420 ... 3560110324

2610124554 ... 63432034052123240342 ... 3321325341

4/6

Figure 3.9: Robby, the soda-can-collection robot – resulting Java Applet

Robby when performing any strategy. The last part – theory visualization – placed on the bottom of the applet is a completely passive unit used just for demonstrating a theoretical background of the genetic algorithm.

We can further divide these parts into smaller utilities. Following sections list them in turn and elucidate their functionality.

Status

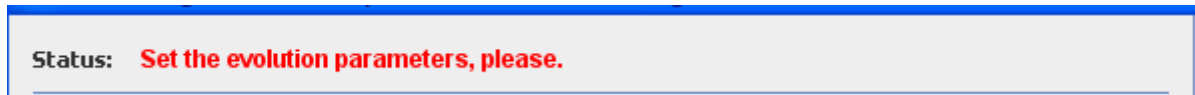


Figure 3.10: "Status" section

This component cannot be left unmentioned, even though it does not directly serve for applet's control. Status line informs about current applet's activity.

Create a world

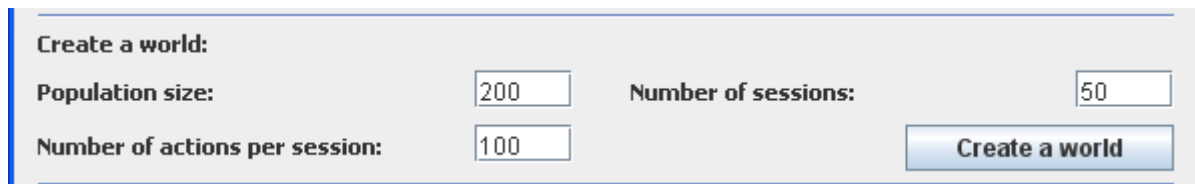


Figure 3.11: "Create a world" section

We decided not to keep a whole genetic algorithm's settings of a piece, because it considerably reduces possibilities of testing. User has to set all parameters of the algorithm in a beginning and eventually does not have a chance to continue in an evolution process. Therefore, we decided to separate parameters into groups. The parameters, that should stay unchanged during whole evolution process, sets user in this component. *Population size*, *Number of sessions* and *Number of actions per session* belong among them. This way, a world, where an evolution can start, is created (see the figure 3.13 on the page 50).

Parameters' upper boundary is set to 4999 (inclusive). It should be sufficient for effective solving Robby's task. *Number of sessions* and *Number of actions per session* have to be an integer greater than 0. *Population size* does not have to be lower than 2. If it was, a crossover operation would not be applied, because two individuals are needed thereto. Default settings is 200 individuals in a population, that are tested in 50 sessions, when each of them counts 50 actions.

Evolution settings

When a world is created, the evolution itself can start. There are two parameters to be set – *Number of generations* and *Mutation probability*, that are preset to 100 and 0.005 as default (respectively). *Number of generations* has to fall into an interval of integers (0, 50 000) and *Mutation probability* is a decimal between 0 and 1 (inclusive). By clicking the button *Evolve*, a loop of an algorithm described in the section 3.2.2 on the pages 46 and 47 is started.

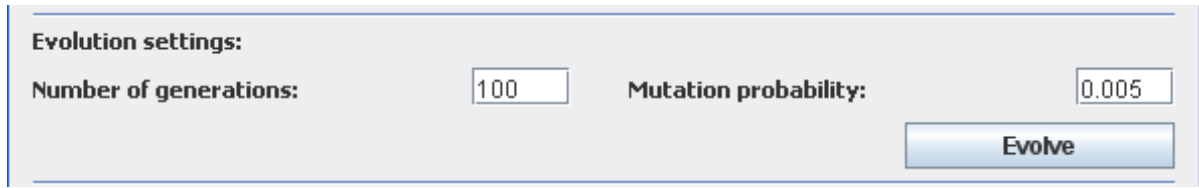


Figure 3.12: "Evolution settings" section

While the computation is finished, user can still continue with setting another value to the parameters and start the algorithm again. Therefore, it is better to start with lower number of generations and eventually increase it afterwards.

Following figure illustrates the idea of dividing the algorithm's parameters into two parts.

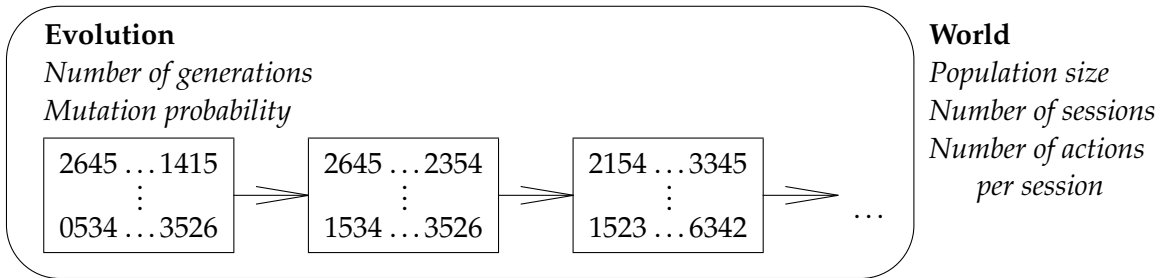


Figure 3.13: Division of the algorithm into two parts – World and Evolution

Evolution results

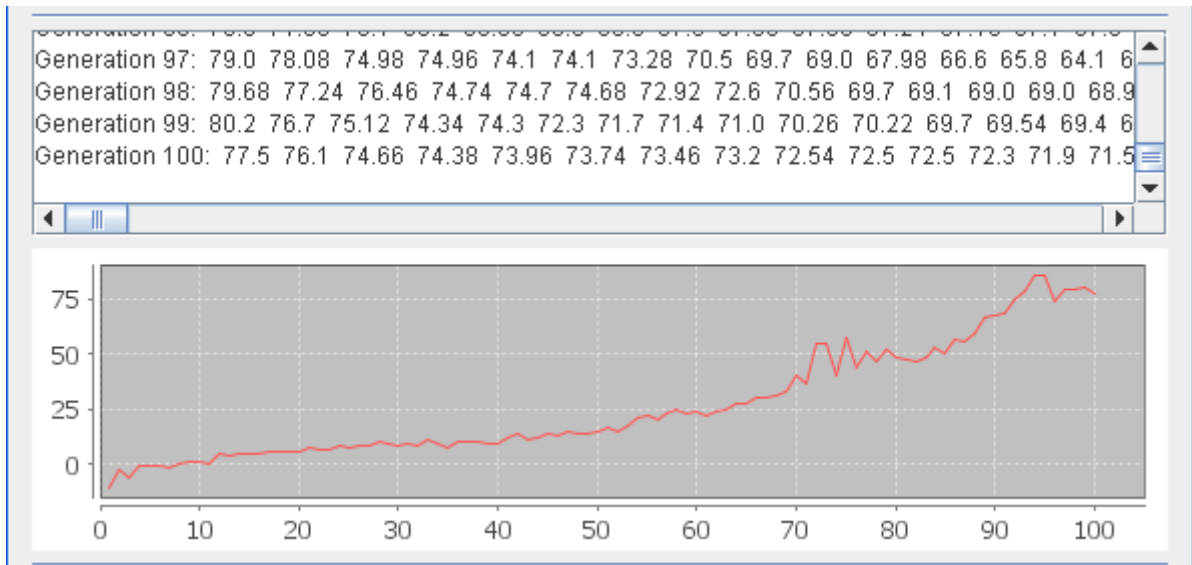


Figure 3.14: "Evolution results" section

The "evolution results" component consists of two parts – a text area and a graph. The text area serves for informing a user about ongoing evolution. Each line represents single generation and individuals within. Values on the line stand for individuals' fitnesses in descending order from left to right. The lines in the text area are added continuously during an evolution.

By contrast, the graph is updated once per evolution cycle – on its end. It represents progress of the best individual’s fitness in a current population. When another evolution cycle in the same world is finished, the graph does not illustrate overall progress, but only a course of a current loop.

Both the units are erased and newly initialized when a new world is created.

Best evolved strategy

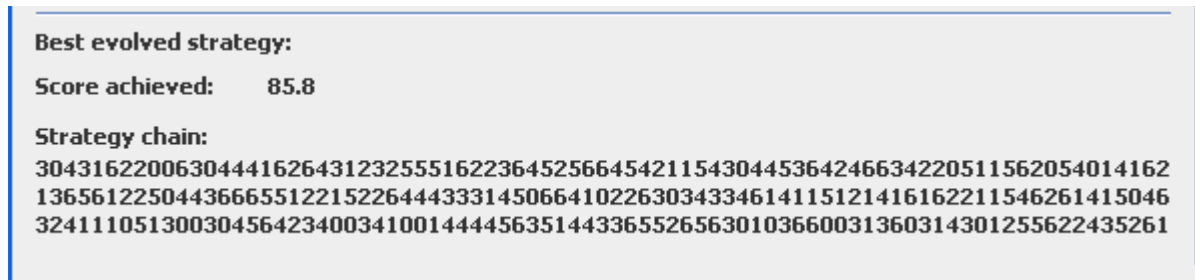


Figure 3.15: “Evolution results” section




This component just completes the results of an evolution cycle by showing the best strategy evolved on a current world. The best strategy can occur not just in the last evolved generation, therefore it is crucial to compare the best strategy in a current generation with the best strategy overall which has been evolved by now.

The best evolved strategy is represented by its fitness value and a strategy chain. The strategy chain consists of 243 integers, that describe Robby’s behaviour in various situations (see the section 3.2.2 for more detailed description). List of the situations is a content of Appendix A on the page 60.

Playground visualization

A next to the last component is not directly involved in an evolution process. It serves for visualizing Robby’s task and consists of 10×10 grid. By clicking the *Create playground* button, a new playground randomly strewn with cans is established. It uses settings of a current world – namely value of the variable *Number of actions per session*. Therefore this component cannot be used before a world is created.

A strategy to be tested is specified in a text field *Strategy chain*. A currently best evolved strategy is automatically inserted when an evolution loop ends. Alternatively it can be added by user (see Appendix A on a page 60 for an order of situations in a strategy chain). When the *Test strategy* button is clicked, a strategy starts being tested. Robby discovers his situation and performs appropriate action. A period of one step is set to one second and *Number of actions per session* actions is performed. It can happen that Robby, according to his strategy, performs repetitively one step (e.g. hitting a wall and bouncing back). Therefore this component contains a *Stop testing procedure* button.

By reason of limited space, we could not use any prettier graphics than simple icons. Robby is figured as a red square: . When hitting a wall and bouncing back, a red line in a direction of Robby’s move is added (Robby is moving south in this case): . And finally a soda can is represented as a small black square: . Another situations are illustrated as a combination of aforementioned icons (and rotated in some cases).

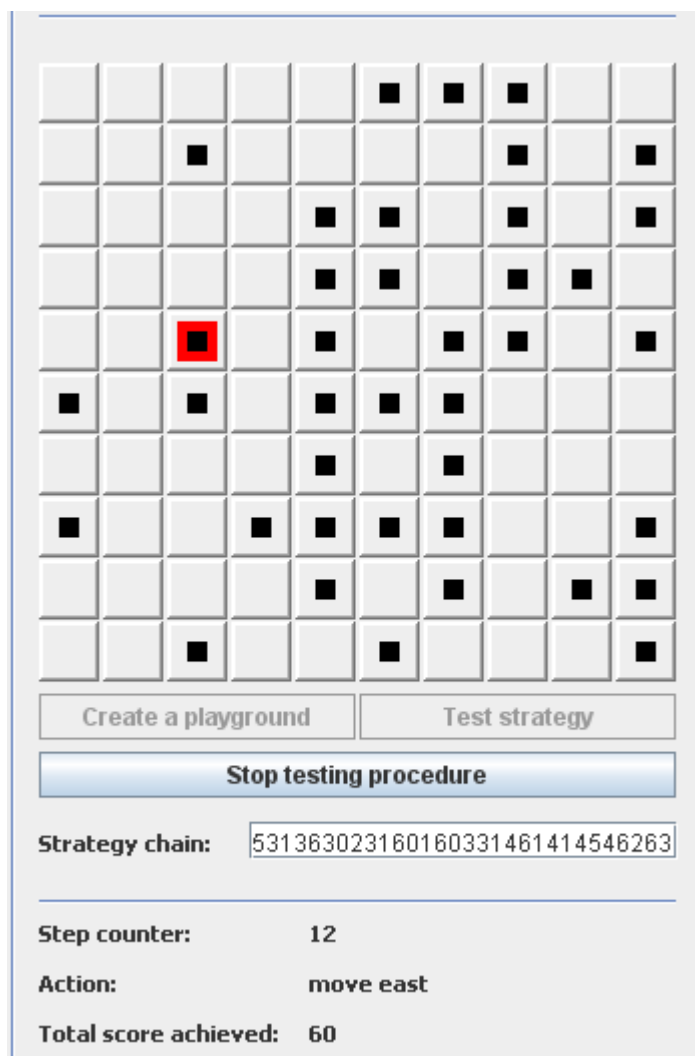


Figure 3.16: "Playground visualization" section

Theory visualization

The evolution process consists of evolving a given number of generations, where strategies contained in each of the contiguous generations try to show better behavior than in the previous one.

Evolving a new generation consists of:

1. Compute a fitness for each strategy in the previous generation to set their ranking.
2. Choose two of the previous generation's strategies.
The higher the fitness is, the higher the chance to be chosen is.
3. Apply a crossover operation on them.
4. Apply a mutation operation on them.
5. Place them into the new generation. Continue from the point 2 (choosing a pair of strategies) until the new generation contains the same number of strategies as the previous.

1/6

Figure 3.17: "Theory visualization" section

The last component occupies bottom part of the applet on its whole width. Its purpose is to fill user's time when a computation runs by giving a user brief description of a genetic algorithm processed inside the applet. When the button *Create world* is clicked, a first part about a strategy representation appears.

When an evolution is started by clicking the button *Evolve*, another part, dealing with a course of genetic algorithm, is shown. It consists of 6 pages that can user scroll through.

3.2.4 Experiments with applet's setup

Another topic is directly connected to the applet's description – experiments. Unfortunately, a volume of this section does not conform to section 2.2.4 dealing with similar subject in the field of neural networks. The main reason is, that a performance of genetic algorithms greatly depends on a random initial generation. Simply speaking, if we were out of luck during a creation of an initial generation and got some unfitting individuals, even the genetic algorithm cannot find the most optimal solution then, because which individuals it gets in the beginning, that only can use in its operations.

Therefore, we present here just a few findings based on our observations. Following paragraphs discuss the applet parameters' effect on the algorithm's performance.

Population size

The first and the most important parameter is a size of a population. The larger the population is, the wider options the algorithm has. Nevertheless, it does not guarantee a good solution. In the beginning, the algorithm works with a large variety of individuals. With increasing number of generations, the algorithm starts to work with smaller and smaller group of individuals with high fitness. It can happen, that another individual outside this group is chosen for reproduction, but it is not highly probable. Therefore, the other individuals stay away and do not contribute to the evolution, despite they would improve a final result. Another drawback of a large population is a time-consuming handling of it.

On the other hand, the evolution of smaller population takes significantly less time, but its success depends more on a composition of an initial generation.

Therefore it is necessary to find such a population size that provides enough free scope and its computation is not highly time-consuming. It is not a simple task and there is no simple solution. It is difficult to say, what a large population is. An individual's string composes of 243 situations. Each of them can be solved in 7 ways. Therefore, the total amount of strategies is $7^{243} = 2.28 \cdot 10^{205}$ and a large population could not be called one, that counts 5 000 and not even 10 000 individuals.

The applet's parameter *Population size* has got a range of (1, 5000). Hence, it is not possible to work with a really large population. But even if the boundary was higher, it would not be more helpful, because computation time of the evolution process would become unbearable.

We suggest to work with smaller populations with a size of 200 – 400 individuals. This size should offer enough free scope and bearable time of computation. It can happen, that initial generation does not have such potential to evolve good solutions. Nevertheless, the computation does not take too long, that we cannot start another try.

We conducted an experiment with three worlds that have same values of all parameters except *Population size*. Values of 1000, 200 and 200 were used for this parameter. As we expected, time of computation was much lower for the worlds with population of 200. Nevertheless, their computation ended on scores 200.1 and 54.8, while the world with population 1000 gained a score 204.

As a result, we can say, that if a computation does not end on lower values, there is no difference in using a population of 200 and 1000. Therefore it is better to use smaller populations and if a computation sticks and does not grow, we can just start again. See the table 3.4 on the following page for an overview.

	World 1	World 2	World 3
Population size	1000	200	200
Number of sessions	50	50	50
Number of actions per session	50	50	50
Number of generations	500	500	500
Mutation probability	0.005	0.005	0.005
Score of a best evolved strategy	204.0	59.8	200.1
Approximate computation time (min)	65	14.5	14.5

Table 3.4: Summary of an experiment with different values of *Population size*

Number of sessions

A value of the second parameter does not influence applet's behavior as much as a size of a population. The higher *Number of sessions* is, the more general meaning the fitness value has. Simply speaking, this parameter conforms to the number of playgrounds, that strategies are tested in. If a high number of various playgrounds is used, the more certainly the tested strategy will not fail in any other. Accurate value is individual and its escalation costs additional time of a computation.

Number of actions per session

Number of actions per session is analogous to the previous parameter. The higher its value is, the more general meaning the fitness function has. If Robby does more actions in a playground, the better his current strategy is proven. It is important to remember, that a playground consists of $10 \times 10 = 100$ sites. We can simply construct a strategy, that goes through all sites one by one in 100 steps and picks up 50 cans¹⁷. Hence its fitness value is 500. If we want to evolve a strategy that shows better results, the strategy has to vector Robby in a way that he picks up the cans faster than in 100 steps. Therefore we prefer, that the parameter *Number of actions per session* is set to a value 100 at most.

Number of generations

Number of generations variable just performs a function of a counter. Its optimal value cannot be set, because a course of the fitness function is unknown in advance. If the fitness value did not grow in last generations, it is highly probable, that it will not grow in future either. Hence, the current best strategy can be considered as a final. See the figure 3.18 for an example.

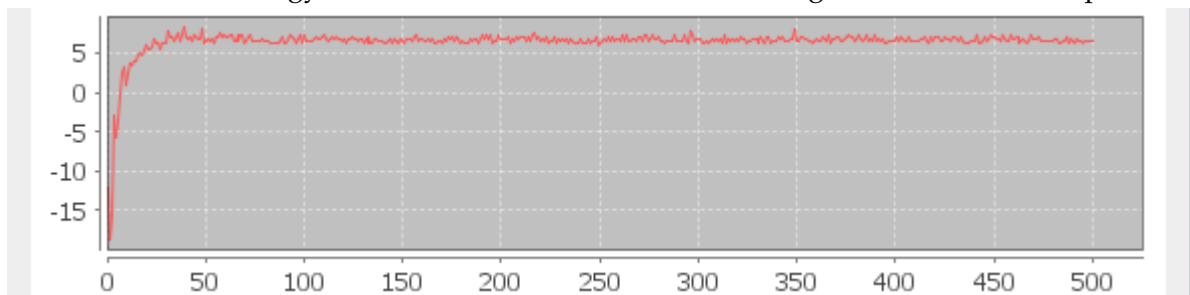


Figure 3.18: Stagnating fitness function

17. Every site has got the probability of 50% that it contains a can. Therefore we can assume in our example, that the number of cans is exactly 50 without limiting the generality.

Mutation probability

A genetic algorithm makes use of two operators – the crossover and the mutation. In general, the first one is applied always and the second rarely. That is because the crossover works with selected pair of strategies and consequently we get offsprings that again consist of a modification of their parents' strings. It is quite predictable process (when omitting the exact placing of a crossover point).

On the other hand, the mutation's result is completely random. Once the mutation is applied, we can never be sure about a result. Therefore it should just serve as a support for crossover and be applied very seldom. The applet's default value of *Mutation probability* is set to 0.005 and should be held very low. See the figure 3.19, where a result of the applet's run with *Mutation probability* equal to 0.1 is illustrated.

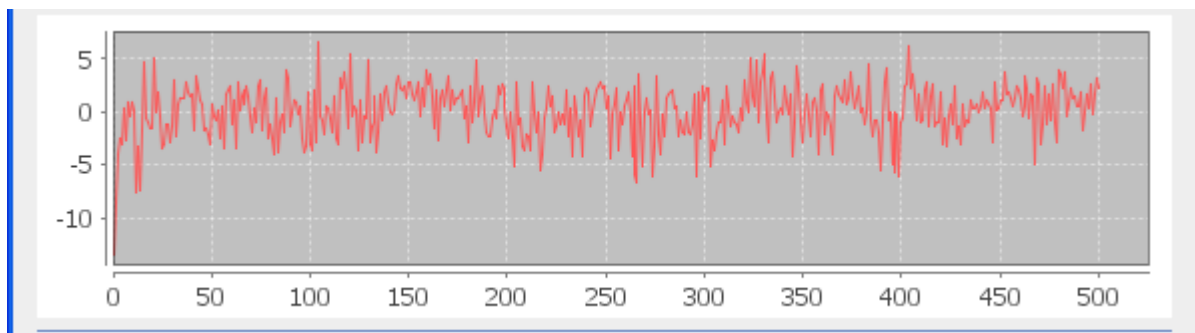


Figure 3.19: Applet run with *Mutation probability* = 0.1

How to evaluate results

At last, we briefly sum up how to interpret the applet's results as a whole. When discussing an influence of the parameter *Number of actions per session*, we presented a simple idea, that it is not very difficult to create a handmade strategy, which goes through all sites of a playground in 100 steps and collects all cans. Therefore an evolved strategy should be better than our handmade piece to be called successful. Of course, it is not possible to evolve strategy, that gains more than 500 points, when there are just 50 cans in a playground. Successful is a strategy that makes it faster. Talking in general, if *Number of actions per session* = < 100 and a strategy's score is greater than

$$\text{Threshold of success} = \frac{\text{Number of actions per session}}{2} \cdot 10 \quad (3.2)$$

we can call it successful. *Threshold of success* represents a value of an aforementioned handmade strategy for *Number of actions per session* steps.

3.2.5 User's guide

As well as the chapter 2, the section devoted to a topic of genetic algorithms ends with a user's guide that should serve as a manual for users, that do not want to study whole theoretical background and just simply use the applet.

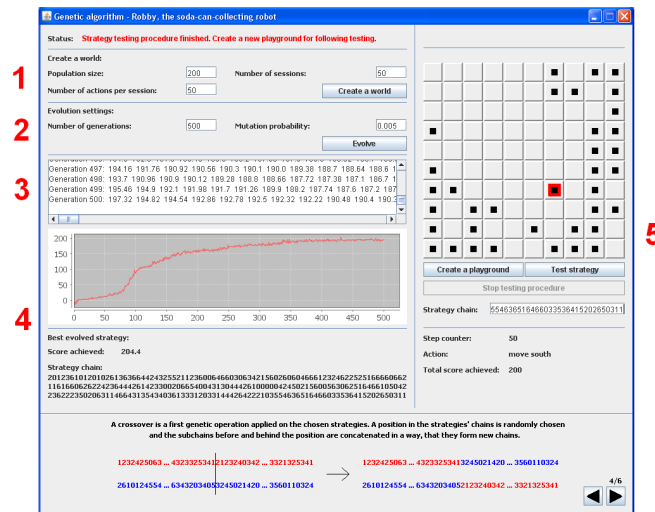


Figure 3.20: User's guide: World – Evolution – Interim results – Best strategy – Playground

There is a *Status* line at the top of the applet. Please notice the line, it informs about current applet's activity.

When the applet is working, there is a presentation of genetic algorithm's theory running in its lowermost part. It changes according to a currently running task. You can turn pages by clicking arrow buttons.

1. Create a new world by setting parameters *Population size*, *Number of sessions* and *Number of actions per session*. Their default values are 200, 50 and 50 (respectively). Do not set *Population size* > 1000, running time of a computation rapidly increases then. Keep remaining two parameters less than or equal to 100. These settings are binding for all rounds of evolution run in this world. Complete the creation process by clicking a button *Create a world*.
2. Set evolution parameters *Number of generations* and *Mutation probability*. The more generations you set, the longer the evolution takes. It is better to start with lower values and run the evolution more times, if necessary. Leave the *Mutation probability* less than 0.01. The higher its value is, the more randomly the evolution proceeds and does not lead to desired result. Click a button *Evolve* to run the evolution.
3. The process of evolution can take some time. Check interim results published in a text area. Its one line conforms to one generation. Fitness values of strategies within the generation are sorted from the greatest to the lowest in a direction from left to right.
4. When the evolution is finished, see its results. There is a *graph*, that figures changes of best strategy's fitness value in time. The best strategy overall is presented under the graph.
5. The current best strategy is uploaded into a playground visualization in the right part of the applet. Click the button *Create a playground* to initialize the playground. By clicking *Test strategy*, Robby starts to move in accordance with uploaded strategy. If he is not stopped by a button *Stop testing procedure*, Robby performs number of steps determined by parameter *Number of actions per session*.

Try another parameters' settings and observe how the results of evolution change.

Chapter 4

Conclusion

We aimed to develop user-friendly applications, that would introduce an interesting area of computer science – *bio-inspired computing*. We focused mainly on two of its methods – *neural networks* and *genetic algorithms*. Each of the applications solves a task whereon capabilities of aforementioned methods are easy to see. Neural networks approach was implemented to solve a problem of *language recognition of a text*. And a task of *Robby, the soda-can-collecting robot* makes use of genetic algorithms. Both the applications are presented in the form of Java applets.

It can be looked upon the thesis as two separated parts. Both of them are further divided into halves. The first is devoted to a presentation of theoretical background that is consequently used in the second part describing an application itself.

Chapter 2 deals with a topic of neural networks. According to the aforementioned schema, section 2.1 treats of biological inspiration and a history of neural networks. And continues with a mathematical description of one of their variations – backpropagation. Section 2.2 follows with a detailed explanation of the language recognition application. We conducted a few experiments to test the applet’s capabilities as well.

Chapter 3 treats of a field of genetic algorithms. Section 3.1 is devoted to an inspiration and a history of the topic. Further, a genetic algorithm is specified to fit our purposes. Robby, the soda-can-collecting robot is a name of a task solved in a section 3.2. A detailed description of a result is presented there too. As well as in the chapter 2, we present outcomes of a few experiments conducted on the applet.

We used an object-oriented programming language *Java*. We placed an emphasis on easy and intuitive control of the applets. They bring to users a possibility to study changes in a behaviour of the algorithms by altering their parameters.

The whole project is located on the web page:

<http://is.muni.cz/www/139613/index.html>

Along with Java applets, there is a brief introduction into the problematics and a user’s guide. The text of the thesis is a part of the web page as well.

Bibliography

- [1] Baev, Konstantin V.: *Biological neural networks: Hierarchical concept of brain functions*. Boston, Birkhäuser, 1998.
- [2] McCulloch, Warren S., Pitts, Walter: *A logical calculus of the ideas immanent in nervous activity*. Bulletin of mathematical biophysics, Vol. 5, 1943, pp. 115–133.
- [3] Hebb, Donald O.: *The Organization of behavior: A neuropsychological theory*. New York, Wiley, 1949.
- [4] Rosenblatt, Frank: *A probabilistic model for information storage and organization in the brain*. Psychological review, Vol. 65, 1958, pp. 386–408.
- [5] Widrow, Bernard: *Generalization and information storage in networks of Adaline "neurons"*. Self-organizing Systems, Washington, D.C, Spartan Books, 1962, pp. 435–461.
- [6] Minsky, Marvin L., Papert, Seymour: *Perceptrons*. Cambridge MA, MIT Press, 1969.
- [7] Hopfield, John J.: *Neural networks and physical systems with emergent collective computational abilities*. Proceedings of the national academy of sciences, Vol. 79, 1982, pp. 2554–2558.
- [8] Rumelhart, David E., Hilton, Geoffrey E., Williams, Ronald J.: *Learning internal representations by error propagation*. Parallel distributed processing: explorations in the microstructure of cognition, Vol. 1, Cambridge MA, MIT Press, 1986, pp. 318–362.
- [9] Bryson, Arthur E., Ho, Yu-Chi: *Applied optimal control*. New York, Blaisdell, 1969.
- [10] Šíma, Jiří, Neruda, Roman: *Teoretické otázky neuronových sítí*. Prague, MATFYZPRESS, 1996.
- [11] Fogel, Lawrence J., Owens, Alvin J., Walsh, Michael J.: *Artificial intelligence through simulated evolution*. New York, Wiley, 1966.
- [12] Holland, John H.: *Adaptation in natural and artificial systems*. Ann Arbor, University of Michigan Press, 1975.
- [13] *TalkOrigins archive: Exploring the creation/Evolution controversy* [online]. Last actualization unknown [cit. 2010-05-06]. Available on the address: <http://www.talkorigins.org/>.
- [14] Flake, Gary W.: *The computational beauty of nature*. Cambridge, Massachusetts, The MIT Press, 1998.
- [15] Pfeifer, Rolf, Bongard, Josh: *How the body shapes the way we think*. Cambridge, Massachusetts, The MIT Press, 2006.

- [16] *CS 441/541: Artificial Intelligence* [online].
Last actualization unknown [cit. 2010-05-14]. Available on the address:
<http://web.cecs.pdx.edu/~mm/ArtificialIntelligenceFall2008/index.html>.

- [17] *CS 441/541: Artificial Intelligence – Homework 6* [online].
Last actualization 2008-11-10 [cit. 2010-05-14]. Available on the address:
<http://www.cs.pdx.edu/~mm/ArtificialIntelligenceFall2008/Homework/Homework6.pdf>.

- [18] Mitchell, Melanie: *Complexity: A guided tour*. New York, Oxford university press, 2009, p. 130-142.

- [19] *Java Platform SE 6 – API* [online].
Last actualization unknown [cit. 2010-05-26]. Available on the address:
<http://java.sun.com/javase/6/docs/api/>.

Appendix A

List of Robby's situations

Table A.1 defines meaning of positions in a strategy's string. Content of one row means following:

| Position in a string | North South East West Current site |

Location in the string is placed in the first column. The second column characterizes Robby's situation (his surroundings and current site).

	Situation		Situation
1	Empty Empty Empty Empty Empty	31	Empty Empty Empty Wall Wall
2	Can Empty Empty Empty Empty	32	Wall Can Empty Empty Empty
3	Empty Can Empty Empty Empty	33	Wall Empty Can Empty Empty
4	Empty Empty Can Empty Empty	34	Wall Empty Empty Can Empty
5	Empty Empty Empty Can Empty	35	Wall Empty Empty Empty Can
6	Empty Empty Empty Empty Can	36	Empty Wall Can Empty Empty
7	Wall Empty Empty Empty Empty	37	Empty Wall Empty Can Empty
8	Empty Wall Empty Empty Empty	38	Empty Wall Empty Empty Can
9	Empty Empty Wall Empty Empty	39	Empty Empty Wall Can Empty
10	Empty Empty Empty Wall Empty	40	Empty Empty Wall Empty Can
11	Empty Empty Empty Empty Wall	41	Empty Empty Empty Wall Can
12	Can Can Empty Empty Empty	42	Can Wall Empty Empty Empty
13	Can Empty Can Empty Empty	43	Can Empty Wall Empty Empty
14	Can Empty Empty Can Empty	44	Can Empty Empty Wall Empty
15	Can Empty Empty Empty Can	45	Can Empty Empty Empty Wall
16	Empty Can Can Empty Empty	46	Empty Can Wall Empty Empty
17	Empty Can Empty Can Empty	47	Empty Can Empty Wall Empty
18	Empty Can Empty Empty Can	48	Empty Can Empty Empty Wall
19	Empty Empty Can Can Empty	49	Empty Empty Can Wall Empty
20	Empty Empty Can Empty Can	50	Empty Empty Can Empty Wall
21	Empty Empty Empty Can Can	51	Empty Empty Empty Can Wall
22	Wall Wall Empty Empty Empty	52	Can Can Can Empty Empty
23	Wall Empty Wall Empty Empty	53	Can Can Empty Can Empty
24	Wall Empty Empty Wall Empty	54	Can Can Empty Empty Can
25	Wall Empty Empty Empty Wall	55	Can Empty Can Can Empty
26	Empty Wall Wall Empty Empty	56	Can Empty Can Empty Can
27	Empty Wall Empty Wall Empty	57	Can Empty Empty Can Can
28	Empty Wall Empty Empty Wall	58	Empty Can Can Can Empty
29	Empty Empty Wall Wall Empty	59	Empty Can Can Empty Can
30	Empty Empty Wall Empty Wall	60	Empty Can Empty Can Can

A. LIST OF ROBBY'S SITUATIONS

Situation		Situation	
61	Empty Empty Can Can Can	107	Wall Empty Empty Can Wall
62	Can Can Wall Empty Empty	108	Empty Wall Can Wall Empty
63	Can Can Empty Wall Empty	109	Empty Wall Can Empty Wall
64	Can Can Empty Empty Wall	110	Empty Wall Empty Can Wall
65	Can Empty Can Wall Empty	111	Empty Empty Wall Can Wall
66	Can Empty Can Empty Wall	112	Can Wall Wall Empty Empty
67	Can Empty Empty Can Wall	113	Can Wall Empty Wall Empty
68	Empty Can Can Wall Empty	114	Can Wall Empty Empty Wall
69	Empty Can Can Empty Wall	115	Can Empty Wall Wall Empty
70	Empty Can Empty Can Wall	116	Can Empty Wall Empty Wall
71	Empty Empty Can Can Wall	117	Can Empty Empty Wall Wall
72	Can Wall Can Empty Empty	118	Empty Can Wall Wall Empty
73	Can Wall Empty Can Empty	119	Empty Can Wall Empty Wall
74	Can Wall Empty Empty Can	120	Empty Can Empty Wall Wall
75	Can Empty Wall Can Empty	121	Empty Empty Can Wall Wall
76	Can Empty Wall Empty Can	122	Wall Wall Wall Empty Empty
77	Can Empty Empty Wall Can	123	Wall Wall Empty Wall Empty
78	Empty Can Wall Can Empty	124	Wall Wall Empty Empty Wall
79	Empty Can Wall Empty Can	125	Wall Empty Wall Wall Empty
80	Empty Can Empty Wall Can	126	Wall Empty Wall Empty Wall
81	Empty Empty Can Wall Can	127	Wall Empty Empty Wall Wall
82	Wall Can Can Empty Empty	128	Empty Wall Wall Wall Empty
83	Wall Can Empty Can Empty	129	Empty Wall Wall Empty Wall
84	Wall Can Empty Empty Can	130	Empty Wall Empty Wall Wall
85	Wall Empty Can Can Empty	131	Empty Empty Wall Wall Wall
86	Wall Empty Can Empty Can	132	Can Can Can Can Empty
87	Wall Empty Empty Can Can	133	Can Can Can Empty Can
88	Empty Wall Can Can Empty	134	Can Can Empty Can Can
89	Empty Wall Can Empty Can	135	Can Empty Can Can Can
90	Empty Wall Empty Can Can	136	Empty Can Can Can Can
91	Empty Empty Wall Can Can	137	Can Can Can Wall Empty
92	Wall Wall Can Empty Empty	138	Can Can Can Empty Wall
93	Wall Wall Empty Can Empty	139	Can Can Empty Can Wall
94	Wall Wall Empty Empty Can	140	Can Empty Can Can Wall
95	Wall Empty Wall Can Empty	141	Empty Can Can Can Wall
96	Wall Empty Wall Empty Can	142	Can Can Wall Can Empty
97	Wall Empty Empty Wall Can	143	Can Can Wall Empty Can
98	Empty Wall Wall Can Empty	144	Can Can Empty Wall Can
99	Empty Wall Wall Empty Can	145	Can Empty Can Wall Can
100	Empty Wall Empty Wall Can	146	Empty Can Can Wall Can
101	Empty Empty Wall Wall Can	147	Can Wall Can Can Empty
102	Wall Can Wall Empty Empty	148	Can Wall Can Empty Can
103	Wall Can Empty Wall Empty	149	Can Wall Empty Can Can
104	Wall Can Empty Empty Wall	150	Can Empty Wall Can Can
105	Wall Empty Can Wall Empty	151	Empty Can Wall Can Can
106	Wall Empty Can Empty Wall	152	Wall Can Can Can Empty

A. LIST OF ROBBY'S SITUATIONS

	Situation		Situation
153	Wall Can Can Empty Can	199	Wall Wall Empty Can Wall
154	Wall Can Empty Can Can	200	Wall Empty Wall Can Wall
155	Wall Empty Can Can Can	201	Empty Wall Wall Can Wall
156	Empty Wall Can Can Can	202	Wall Wall Wall Can Empty
157	Can Can Wall Wall Empty	203	Wall Wall Wall Empty Can
158	Can Can Wall Empty Wall	204	Wall Wall Empty Wall Can
159	Can Can Empty Wall Wall	205	Wall Empty Wall Wall Can
160	Can Empty Can Wall Wall	206	Empty Wall Wall Wall Can
161	Empty Can Can Wall Wall	207	Wall Wall Wall Wall Empty
162	Can Wall Can Wall Empty	208	Wall Wall Wall Empty Wall
163	Can Wall Can Empty Wall	209	Wall Wall Empty Wall Wall
164	Can Wall Empty Can Wall	210	Wall Empty Wall Wall Wall
165	Can Empty Wall Can Wall	211	Empty Wall Wall Wall Wall
166	Empty Can Wall Can Wall	212	Can Can Can Can Can
167	Wall Can Can Wall Empty	213	Can Can Can Can Wall
168	Wall Can Can Empty Wall	214	Can Can Can Wall Can
169	Wall Can Empty Can Wall	215	Can Can Wall Can Can
170	Wall Empty Can Can Wall	216	Can Wall Can Can Can
171	Empty Wall Can Can Wall	217	Wall Can Can Can Can
172	Can Wall Wall Can Empty	218	Can Can Can Wall Wall
173	Can Wall Wall Empty Can	219	Can Can Wall Can Wall
174	Can Wall Empty Wall Can	220	Can Wall Can Can Wall
175	Can Empty Wall Wall Can	221	Wall Can Can Can Wall
176	Empty Can Wall Wall Can	222	Can Can Wall Wall Can
177	Wall Can Wall Can Empty	223	Can Wall Can Wall Can
178	Wall Can Wall Empty Can	224	Wall Can Can Wall Can
179	Wall Can Empty Wall Can	225	Can Wall Wall Can Can
180	Wall Empty Can Wall Can	226	Wall Can Wall Can Can
181	Empty Wall Can Wall Can	227	Wall Wall Can Can Can
182	Wall Wall Can Can Empty	228	Can Can Wall Wall Wall
183	Wall Wall Can Empty Can	229	Can Wall Can Wall Wall
184	Wall Wall Empty Can Can	230	Wall Can Can Wall Wall
185	Wall Empty Wall Can Can	231	Can Wall Wall Can Wall
186	Empty Wall Wall Can Can	232	Wall Can Wall Can Wall
187	Can Wall Wall Wall Empty	233	Wall Wall Can Can Wall
188	Can Wall Wall Empty Wall	234	Can Wall Wall Wall Wall
189	Can Wall Empty Wall Wall	235	Wall Can Wall Wall Can
190	Can Empty Wall Wall Wall	236	Wall Wall Can Wall Can
191	Empty Can Wall Wall Wall	237	Wall Wall Wall Can Can
192	Wall Can Wall Wall Empty	238	Can Wall Wall Wall Wall
193	Wall Can Wall Empty Wall	239	Wall Can Wall Wall Wall
194	Wall Can Empty Wall Wall	240	Wall Wall Can Wall Wall
195	Wall Empty Can Wall Wall	241	Wall Wall Wall Can Wall
196	Empty Wall Can Wall Wall	242	Wall Wall Wall Wall Can
197	Wall Wall Can Wall Empty	243	Wall Wall Wall Wall Wall
198	Wall Wall Can Empty Wall		

Table A.1: List of situations

Appendix B

Java applet deployment

Although an applet deployment looks like a routine task, sometimes it can cause a lot of problems. Therefore we present here the procedure that was used for our applets. Following lines describe its application on the language recognition applet.

A Java platform includes a feature called *deployment toolkit* since the Standard Edition 6, update 10. The deployment toolkit is a set of JavaScript functions that can be helpful when deploying applets across various browsers and operating systems. It seems to a user as a simple button that starts the application.



Figure B.1: Java start button

Let's have a little deeper look at what happens when clicking the button (the whole process is illustrated on the figure B.2). The button itself is represented in the HTML code as follows:

```
1 <script src="http://java.com/js/deployJava.js"></script>
2 <script>
3   var url="http://is.muni.cz/www/139613/launch.jnlp"
4       deployJava.createWebStartLaunchButton(url, "1.6")
5 </script>
```

The deployment toolkit is loaded on the line 1 and consequently the function called `createWebStartLaunchButton` creates the button on a current page. The `url` variable defines a location of a *JNLP file*¹. The second parameter defines a minimal version of the Java environment required for running the applet.

Nevertheless, the more important task of the `createWebStartLaunchButton` function is to start the application when the launch button is clicked. It is realized by calling the function `launchWebStartApplication` with a parameter, that specifies the location of the JNLP file.

JNLP file defines how to launch the applet. It contains a XML schema that specifies information about the applet, a location of jar files, required java version, etc.

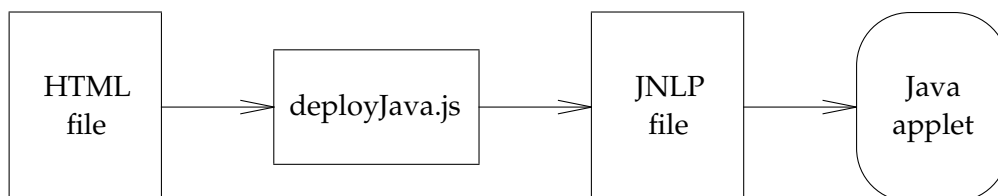


Figure B.2: Deployment procedure when using Java deployment kit

1. *JNLP* – Java Network Launching Protocol

```

1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <jnlp codebase="http://is.muni.cz/www/139613/" href="launch.jnlp"
   spec="1.0+">
3    <information>
4      <title>Neural network - language recognition</title>
5      <vendor>David Kabáth</vendor>
6      <homepage href="http://is.muni.cz/www/139613/" />
7      <description>Neural network - language recognition</description>
8      <description kind="short">Language recognition based on artificial
       neural network.</description>
9    </information>
10   <security>
11     <all-permissions/>
12   </security>
13   <resources>
14     <j2se version="1.5+" />
15     <jar eager="true" href="neural_networks_finish.jar" main="true" />
16     <jar href="10204676/jcommon-1.0.16.jar" />
17     <jar href="10204676/swing-layout-1.0.4.jar" />
18     <jar href="10204676/jfreechart-1.0.13.jar" />
19   </resources>
20   <applet-desc height="737" width="954" name="neural_networks_finish"
    main-class="nn_backpropagation.nn_backpropagation_applet">
21   </applet-desc>
22 </jnlp>

```

Line 2 specifies the JNLP file itself by codebase and href. Following lines 3–9 contain the description of the applet. Lines 10–12 check and provide the security. In this case, the `<all-permissions/>` option allows the applet to access a user's hard disc. Therefore all jar files used by the applet have to be signed (or self-signed). Subsequently the lines 13–19 locate used jar files and finally the `<applet-desc>` section sets the property of the applet. The most important is the `main-class`, where the name of the main class is specified in the form 'package.class'. Please see the Java tutorials page for more detailed description². At last, the jar files defined in the JNLP file are downloaded and launched.

This procedure may seem to be a little more complicated than an applet deployment provided by a simple HTML tag `<applet>`, but benefits of the deployment kit outweigh its complexity. The greatest advantage is a Java environment checking, whether the applet can be run on the current machine. If not, a user is redirected to the official website, where the required Java version can be downloaded. Hence, it would not happen, that an applet fails to start with an error message.

Among the other advantages we would like to point out the fact, that the sources of the applet are defined in the JNLP file. Therefore, we can refer to more jar files. We used four jar files in our example that are defined on lines 15–18. The first one is an application's file and the others are used libraries. Hence, we do not have to put it all in one jar file, but leave it separated.

Finally, the complicacy of programming a JNLP file can be compensated with the use of any development environment (e.g. NetBeans) that is able to generate jar files as well as a JNLP script.

2. <http://java.sun.com/docs/books/tutorial/deployment/deploymentInDepth/jnlpFileSyntax.html>

Appendix C

Content of the enclosed compact disc

- Text of the thesis – PDF, TEX
- Applets – JAR, NetBeans Projects
- Web pages