BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

CONSTRUCTIVE NEURAL NETWORKS
NEURONOVÉ SÍTĚ S PROMĚNNOU TOPOLOGIÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. TOMÁŠ ČERNÍK
SUPERVISOR VEDOUCÍ PRÁCE	doc. Ing. FRANTIŠEK ZBOŘIL, CSc.

BRNO 2016

Zadání diplomové práce/6929/2015/xcerni08

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Černík Tomáš, Bc.**

Obor: Inteligentní systémy

Téma: **Neuronové sítě s proměnnou topologií**
Constructive Neural Networks

Kategorie: Umělá inteligence

Pokyny:

1. Prostudujte zadanou literaturu.
2. Vypracujte přehled existujících algoritmů pro změnu topologií neuronových sítí.
3. Vyberte alespoň dva z těchto algoritmů a implementujte je.
4. Posuďte možné modifikace/vylepšení vybraných algoritmů.
5. S vybranými/modifikovanými algoritmy proveďte experimenty se zaměřením buď na klasifikaci, nebo heteroasociaci.
6. Zhodnoťte dosažené výsledky.

Literatura:

- Sharma, S. K., Chandra, P.: Constructive Neural Networks: A review, International Journal of Engineering Science and Technology Vol. 2(12), 2010
- Franco, L., Elizondo, D. A., Jerez, J. M. (eds): Constructive neural networks, Springer, 2009, ISBN 978-3-642-04511-0
- Ondráček, T.: Adaptivní vícevrstvé neuronové sítě, disertační práce, FIT VUT v Brně, 2005

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zbořil František V., doc. Ing., CSc.**, UITS FIT VUT

Datum zadání: 1. listopadu 2015

Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

[illegible]

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstract
Master theses deals with Constructive Neural newtorks. First part describes neural networks and coresponding mathematical models. Furher, it shows basic algorithms for learning neural networks and desribes basic constructive algotithms and their modifications. The second part deals with implementation details of selected algorithms and provides their comparision. Further comparision with backpropagation algorithm is provided.

Abstrakt
Tato práce se zabývá neuronovými sítěmi - konkrétně sítěmi s proměnnou topologií. Teoretická část popisuje neuronové sítě a jejich matematické modely. Dále ukazuje základní algoritmy pro učení neuronových sítí a rozebírá několik základních konstruktivních algoritmů a jejich rozšíření. Praktická část se zaobírá implementací vybraných konstruktivních algoritmů a uvádí jejich porovnání. Dále jsou algoritmy srovnány s učícím algoritmem backpropagation.

Keywords
Neural Networks, Soft Computing, Constructive Neural Networks, C++, Recurent Neural Networks, Cascade network

Klíčová slova
Neuronové sítě, Soft Computing, konstruktivní neuronové sítě, C++, rekurentní neuronové sítě, Kaskádová síť

Reference
ČERNÍK, Tomáš. *Constructive Neural Networks.* Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Zbořil František.

Constructive Neural Networks

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of doc. Ing. František Zbořil, CSc. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Tomáš Černík
May 23, 2016

Acknowledgements

I would like to thank doc. Ing. František Zbořil, CSc., my master's thesis supervisor, for his great guidance and valuable advice, he helped to complete this work. I would also like to thank to my family for their support during my studies.

Contents

List of Figures

Chapter 1

Introduction

These days, neural networks are being used in many applications. In reality, the usage for neural networks can be found in many technical sectors. We can use them in data classification and filtering such as Email filter. They can also be found in games whereby they act as our opponents when playing for example poker, classical ping–pong or chess. Such opponents can adapt to our game style in order to achieve better results.

Neural networks can also be used in health care. Hybrid lung cancer detection system named “HLND”[2] being based on an artificial neural network improves the accuracy of diagnosis and the speed of lung cancer radiology.

Neural networks are a part of methods called **soft computing**, which is an alternative way of approaching mathematical or computational problems. They are suitable for solving many problems, where brute force algorithms fail. **NP-complete** problems or problems with a wide variety of possible solutions could be given as a good example. In many cases the goal of these methods is not to find the best solution – sometimes the best solution is not known, but to find out a solution that is satisfactory. Moreover, neural networks can be used when noisy or incomplete data appear on input – they are able to ignore imprecisions and “guess” the rest of information.

As neural networks are becoming more and more common, a desire to be able to train them quickly as well as scale them is arising. In the course of time, many approaches to learning and building them have been developed. With increasing power of graphical accelerators, the time necessary for computing and training of neural networks is getting shorter. The speedup gained through transition from processors to graphical accelerators has enabled learning to be realized within few hours instead of days. This speedup allows to train networks faster, but the problem of selecting proper size still persists.

This thesis deals with algorithms for development of topology of neural networks. These algorithms try to set the correct size and topology for selected problem to prevent common deficients of neural networks – over-fitting or the inability to learn. The task of this thesis is to examine current algorithms, propose possible improvements, implement them and evaluate their performance on selected experiments.

1.1 Content of thesis

This thesis is subdivided into two logical parts. The first part deals with the theoretical background of neural networks, training algorithms, constructive algorithms and genetic programming. The second part consists of implementation of selected algorithms. Further, possible improvements are proposed and experimentally evaluated.

Chapter 2 describes artificial neural networks and possible classifications of neural networks according to their topology. Chapter 3 describes standard algorithms for training neural networks - Backpropagation, Perceptron learning algorithm and its modification Thermal perceptron learning algorithm. Chapter 4 deals with algorithms for topology development – constructive algorithms. These algorithms are divided into 3 categories – constructive, pruning and neuroevolutionary algorithms. Chapter 5 describes genetic programming, its operators and initialization methods.

Chapter 6 deals with specific implementation of selected algorithms and genetic programming, which is used for development of code trees for neuroevolutionary algorithm. Chapter 7 proposes possible improvements to implemented algorithms. These improvements and original algorithms are compared in chapter 8 and finally evaluation of experiments is presented in chapter 9.

Chapter 2

Artificial Neural Networks

An artificial neural network (**ANN**) is inspired by a biological neural network we know from our brains. **ANN** is a set of artificial neurons and connections.

ANNs have been developed over the years and many types of **ANN**s have been invented. These neural networks are different in many aspects. The difference can be found in topology, type of activation or basis function or in a type of learning.

Another but not less important feature of **ANN**s is their Turing completeness. That means they are able to compute all algorithms - as a classical computer. This fact was proved by Franklin and Garzon, who have created Turing machine with an **ANN** [11].

2.1 Artificial neuron

Artificial neuron is a simplified model of a biological neuron. Artificial neurons are main units in an artificial neural network. As the biological neuron, it has only one output (**Axon**) and many inputs (**Dendrits**). "Body" of a neuron - in biology called **Soma** is described by **basis function**.

The biggest difference between biological and artificial neuron is in the way neurons "fire" their outputs. Artificial neurons are usually computed in terms of a discrete simulation. In every step, the output of neuron is computed. On the other hand – biological neuron fires when it has enough power (with every incoming input it consumes power and when this gained power gets over value, it fires), this process is known as a continuous simulation.

The first mathematical description of neuron was released in 1943 in the work of Warren McCulloch and Walter Pitts [12]. It was a very simple model, consisting of a linear combination of inputs, a step function and only binary inputs and outputs. This model is known as **McCulloch–Pitts (MCP) neuron**.

In this thesis neuron is defined more flexibly, as shown in figure 2.1.

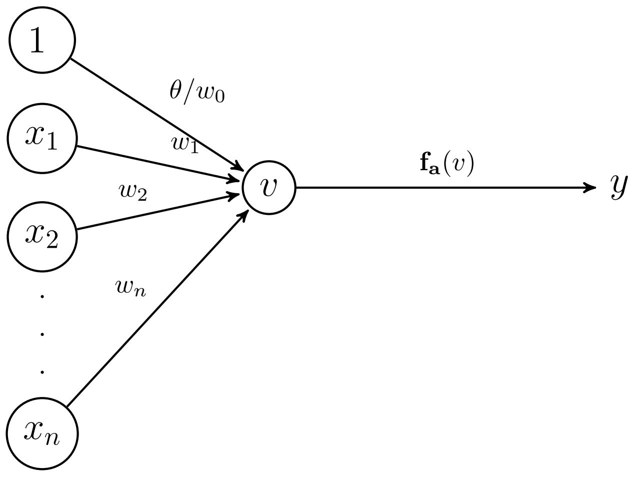


Figure 2.1: Artificial neuron

In figure 2.1 array of neuron's inputs $\vec{x}$ can be seen. Every input has its weight. Such neuron can be described by the following equation.

$$y = \mathbf{f_a}(\mathbf{f_b}(\vec{x}, \vec{w}))$$

Neuron consists of two functions – an activation function and a basis function.

Function ($\mathbf{f_b}$) is known as the basis function. This function describes how to combine inputs and weights and how to compute the value (also known as a potential) of neuron. This is described by the following equation.

$$v = \mathbf{f_b}(\vec{x}, \vec{w})$$

Activation function ($\mathbf{f_a}$) computes output from neuron's value. Often nonlinear function is chosen and is described by the following equation:

$$y = \mathbf{f_a}(v)$$

The activation function can be chosen for its specific behavior. If neural network should approximate **non-linear** function, activation functions of neurons must be chosen from **non-linear** functions. The overview of well known activation and basis functions is introduced further.

Well known **basis functions** are:

- **Linear basis combination (LBF)**

$$\mathbf{f_b}(\vec{x}, \vec{w}) = \sum_{n=1}^{N} x_n \cdot w_n$$

- **Radial basis function (RBF)**

$$\mathbf{f_b}(\vec{x}, \vec{w}) = \sqrt{\sum_{n=1}^{N} (w_n - x_n)^2}$$

Well known **activation functions** are:

- **Gaussian function**

$$\mathbf{f_a}(x) = ae\frac{-(x-b)^2}{2c^2}$$

- **Heaviside (Step function)**

$$\mathbf{f_a}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- **Linear function**

$$\mathbf{f_a}(x) = ax + b$$

- **Logistic function**

$$\mathbf{f_a}(x) = \frac{1}{1+e^{-x}}$$

- **Threshold function** is a generalization of Heaviside

$$\mathbf{f_a}(x) = \begin{cases} 1 & \text{if } x \geq \sigma \\ 0 & \text{otherwise} \end{cases}$$

2.2 Classification of neural networks

Neural networks can be classified from many points of view, for example by the number of layers, topology and application. Every type has its special features. Important classifications for this thesis are:

2.2.1 Classification according to topology

Feed-Forward network is a neural network, where every neuron output is connected only to inputs of neurons in the successive layers. That means there is no cycle in network. These networks are easier to learn and only one iteration is needed to get their output, but they do not have a "memory".

Well known feed-forward networks are **fully-connected**, in which the output of every neuron from previous layer is connected to input of every neuron in the successive layer [21]. We can evaluate this network by computing one layer after another starting from the input layer. In figure 2.2 a fully-connected feed-forward network is depicted.

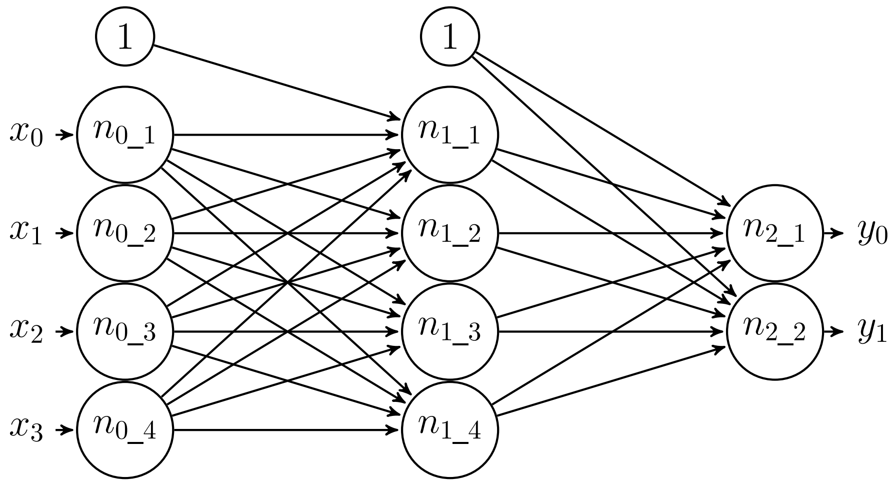


Figure 2.2: Feed forward network

Recurrent network is a neural network with at least one cycle. In this type of network there are no layers. An advantage of this network is its abilities to memorize.

Due to recurrent connections, it is not possible to compute network as simply as feed-forward network. These networks are evaluated iteratively – in the first step the value of each neuron is computed whereas in the second we get their outputs.

These networks are well suitable for problems such as prediction, control of system or speech recognition. However there is still one problem – it is hard to train them and it consumes a lot of time.

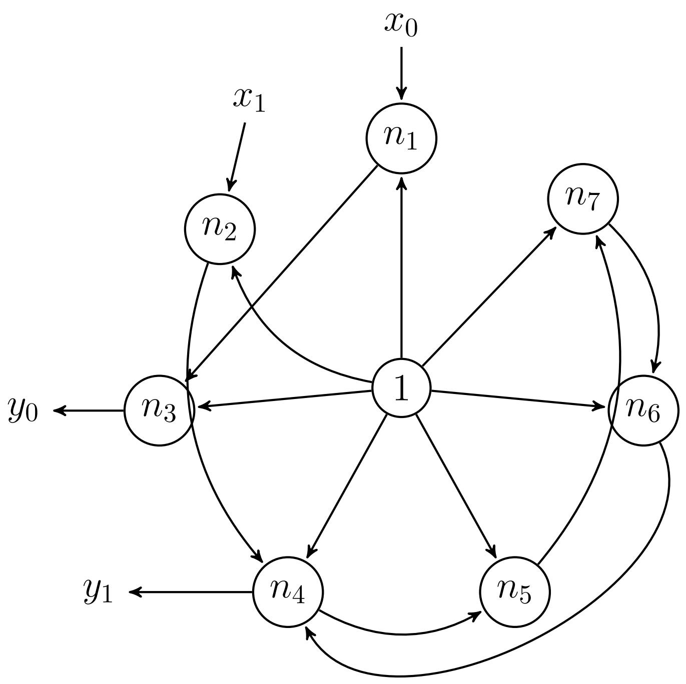


Figure 2.3: Recurrent neural network

2.2.2 Classification according to number of layers

One layer networks are networks which are made of only one layer – the layer is input and also output of network. Example of this network is **Perceptron**.

Many layer networks are networks which consist of more than one layer - we call this layers input, output and hidden according to their position in network.

2.3 Topologies and computation

In this section we are going to describe well known networks and their computation models.

2.3.1 Perceptron

Perceptron was one of the first artificial neural networks. It is a special type of a single-layer feed-forward network consisting of **perceptrons**. Perceptrons are neurons with a step activation function and a linear basis function. It was invented by **Frank Rosenblatt** in 1957[18]. At first, it was built as a machine called **Mark 1 perceptron** and it was designed for an image recognition. This network is very limited due to the fact that it can only learn and approximate linear functions. Its weights were encoded in potentiometers and were updated by electrical motors.

Algorithm 1 describes, how to compute an output for this network.

Algorithm 1 Perceptron

```
procedure PERCEPTRON(x, network)                    ▷ $x$ is input of network
    $y = []$
    for $i = 0; i < network.outputSize; i++$ do
        $valueOfNeuron = 0$
        for $j = 0; j < x.size; j++$ do
            $valueOfNeuron += network.weight[i][j] \cdot x[i]$
                                   ▷ $weight[i][j]$ is weight of neuron $i$ to input $j$
        end for
        $y[i] = valueOfNeuron > 0 ? 1 : 0$
    end for
    return y
end procedure
```

2.3.2 Feed-forward network

Feed-forward network is one of the most common networks used. Further algorithm 2 for computation of feed-forward network is described For simplicity, algorithm describes only **Linear basis function**.

Algorithm 2 Feed-forward Network

```
procedure ComputeFFN(network, input)
    $output = Array[network.numberOfLayers]$
    $output[0] = input$
    for $layer : network.layers$ do
        for $neuron : layer.neurons$ do
            $value = 0$
            for $weight : neuron.weights$ do
                $value = value + outputs[layer - 1][weight] * neuron[weight]$
            end for
            $output[layer][neuron] = f_a(value)$          ▷ $f_a$ is activation value
        end for
    end for
    return $output[network.lastLayer]$
end procedure
```

2.3.3 Recurrent neural network

In this section algorithm 3 for computing new value of fully-connected **RNN** is being provided. This algorithm is more complex than the algorithm for computing **feed-forward network**, because it is necessary to calculate new values for all neurons and then change their outputs. This algorithm is provided for one iteration of computing values. For many problems, output from one iteration is insufficient and many iterations are necessary to get output of network. The number of iterations is usually selected in advance according to specificity of case.

Let us imagine **ANN** playing chess. When an opponent makes a move, **ANN** needs to respond immediately. If only one iteration has been computed, **ANN** could respond to this move after next few rounds and it is too late to make that move – both the board and the situation have changed.

Algorithm 3 Recurrent neural network

```
procedure ComputeRNN(network,input)
    for $neuron : network.inputNeurons$ do
        $neuron.setOutputValue(input[neuron])$          ▷ Set output of input neurons
    end for
    for $neuron : network.neurons$ do          ▷ $f_b$ is basis function of neuron
        $neuron.value = neuron.f_b(network.neurons, neuron.weights)$
    end for
    for $neuron : network.neurons$ do
        $neuron.output = neuron.f_a(neuron.value)$ ▷ $f_a$ is activation function of neuron
    end for
    $output = [\ ]$
    for $neuron : network.outputNeurons$ do
        $output[neuron] = neuron.output$
    end for
    return $output$
end procedure
```

Chapter 3

Algorithms for learning neural networks

The topic of this chapter are basic algorithms for learning artificial neural networks. These algorithms are used by some of constructive algorithms that are introduced later in the thesis.

3.1 Perceptron learning algorithm

This algorithm 4 was developed by Rosenblatt in 1959 [13]. Algorithm modifies weights w_j according to input pattern and difference between the actual computed value and desired output. For simplification only algorithm for two-class classification is mentioned.

Algorithm 4 Perceptron learning algorithm

```
procedure ErrFunction(network,set)
    $Error = 0$
    for all (x,y): set do
        $y' = network.getOutput(x)$
        $Error+ = (y' - y)^2$
    end for
    return $Error/2$
end procedure

procedure Perceptron learning algorithm(network, set:$\{(input, output)\}$)
    $network$ = set weights and thresholds to small random values
    $iterMax$ = user defined value
    $maxError$ = user defined value
    $\eta$ = user defined value from range 0.0-1.0
    while $ErrFunction(network, set) > maxError$ & $iter < iterMax$ do
        $(x, y) = set[iter \mod set.size]$
        $y' = network.getOutput(x)$
        for all w: network.weights do
            $w = w + \eta(y - y')x$
        end for
        iter++
    end while
end procedure
```

3.2 Thermal perceptron learning rule

This algorithm was introduced by Frean M. in 1992[5]. It is a modification of the original **Perceptron learning algorithm** aimed at obtaining a rule that provides stable linearly separable approximation to non-linearly separable problems. It redefines the equation for updating weight.

The original equation $w = w + \eta(y - y')x$ as shown in algorithm 4 is replaced by:

$$w = w + x(y - y')T_{fac}$$

The difference is that the thermal perceptron learning rule incorporates the factor T_{fac}. This value depends on the value of weight and on an artificially introduced temperature T that is decreased as the learning process advances. This technique is commonly used in process called **simulated annealing**. This value can be computed as shown:

$$T_{fac} = \frac{T}{T_0} exp(-\frac{|v|}{T})$$

In this equation, we define T as an actual temperature, v is value of neuron and T_0 is initial temperature set at the beginning of a learning process.

3.3 Backpropagation

Backpropagation is a well known method for supervised learning of neural networks. Backpropagation calculates the gradient of an error function with respect to all weights in the network. Then, the gradient is used by optimization method, which uses computed gradients to update the weights in an attempt to minimize the error of network. Network learned by backpropagation can be a single or multi-layer feed-forward network.[8]

Algorithm can be divided into two parts as shown in algorithm 5. First part consists of computing slopes for changing weights of neuron. The second part update weights to minimize error.

Algorithm 5 Backpropagation

```
 1: procedure BACKPROPAGATE(x,y, network)          ▷ $y$ is desired output, $x$ is input
 2: Part 1. Compute slopes:
 3:     $y' = network.getOutput(x)$                 ▷ $y'$ is output of function
 4:
 5:     Compute $\delta_n$ of every neuron $n$ of output layer according to:
 6:         $\delta_n = (y'[n] - y[n]) \cdot neuron.f'_a(neuron.value)$
 7:                                                 ▷ $f'_a$ is derivation of activation function
 8:
 9:     Compute $\delta_n$ of neurons of hidden layer $L$ starting from layer preceding output layer:
10:         $\delta_n = \sum_j^{\text{neurons of } L+1} (\delta_j \cdot w_{nj}) \cdot neuron.f'_a(neuron.value)$
11:                                                 ▷ $w_{nj}$ is weight between neuron $n$ and $j$
12:
13: Part 2. Compute new weights:
14:     For every hidden and output neuron $n$ of network and its weight $j$ change
15:     weights according to:
16:         $\Delta_{nj} = \alpha \cdot \delta_n \cdot$ (output of neuron that is connected to weight $j$)
17:         $w_{nj} = w_{nj} + \Delta_{nj}$
18:                                                 ▷ $\alpha$ is learning coefficient
19: end procedure
```

Chapter 4

Algorithms for topology development

As **neural networks** were successfully trained by many algorithms, the problem of selecting proper topology by “trial and error” method was still inefficient. This problem involves both choosing the right number of layers and hidden units for layered feed-forward network and selecting proper connections between neurons in recurrent neural network. Over-sized networks with more layers or hidden units are easier to over-fit while smaller networks are not able to learn. This problem mostly affects predictive models, where over-fitting is well seen and neural networks are not able to globalize this process well.

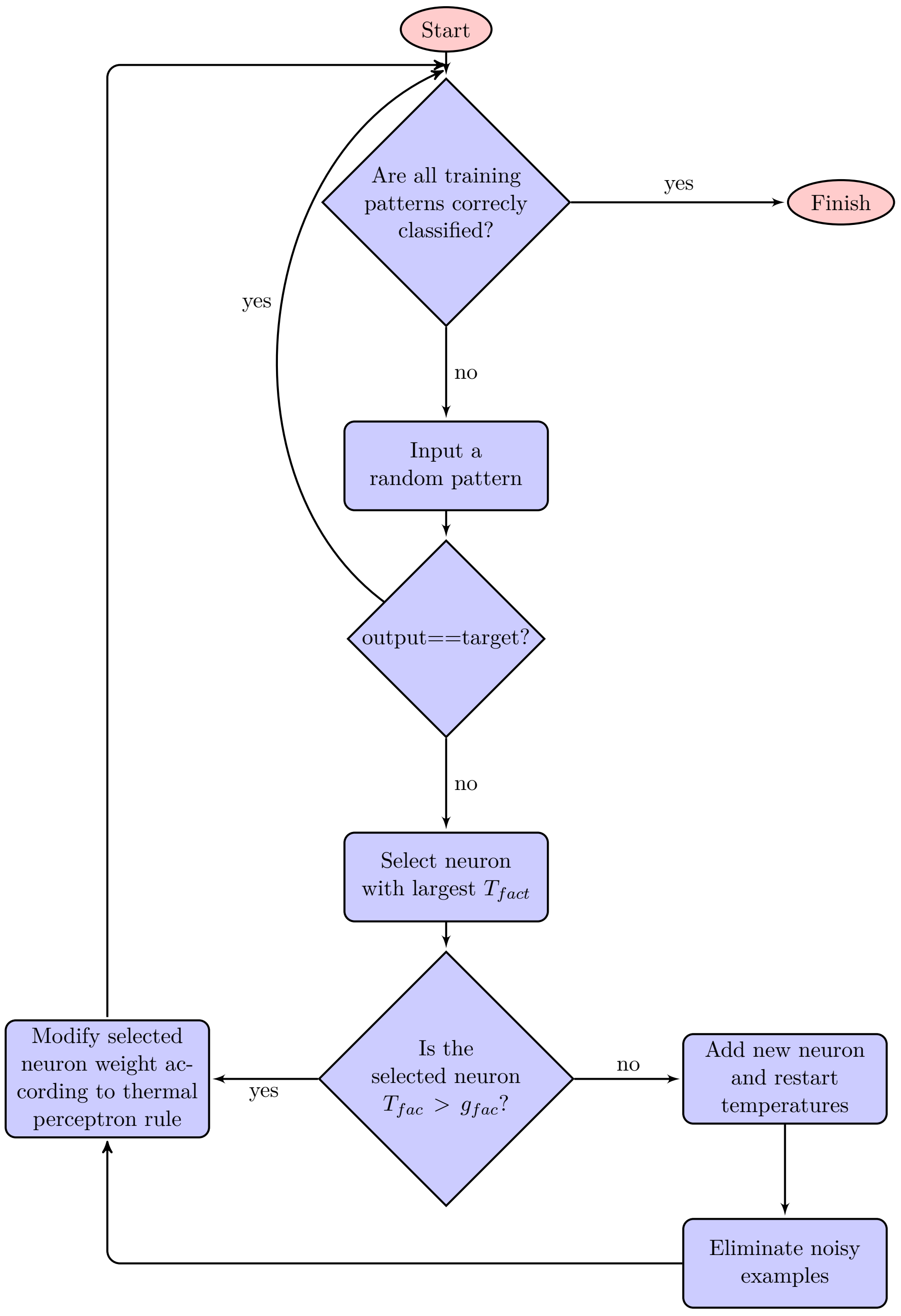
Constructive algorithms can be divided into two categories. The first category consists of algorithms that start with a small network - usually with one hidden unit and work by adding one by one until desired precision is reached. Algorithms from the second category are called **pruning**. These algorithms start with a large network and eliminate unnecessary weights and units one after another.

There is another special group besides these two groups. It is a group consisting of algorithms based on **evolutionary algorithms** and is called neuroevolution.

4.1 C-Mantec algorithm

The **Competitive Majority Network Trained by Error Correction** algorithm creates a structure with a single layer of hidden nodes using step activation function. For a two classes function, it constructs a network with one output neuron computing the “majority function” of the responses of hidden nodes as shown in figure 4.2. That means if more than a half of the hidden neurons is activated the output neuron is activated too. We are going to describe only a two class classifier, but an algorithm for creating multi-class classifier also exists [20].

Figure 4.1: C-Mantec algorithm



Start
Are all training patterns correcly classified?
yes
Finish
no
Input a random pattern
output==target?
yes
no
Select neuron with largest T_{fact}
Is the selected neuron $T_{fac} > g_{fac}$?
yes
Modify selected neuron weight according to thermal perceptron rule
no
Add new neuron and restart temperatures
Eliminate noisy examples

Algorithm starts with a single neuron in a hidden layer and adds more neurons every time when the present ones are not able to classify a whole training set right. Learning is separated into two levels. For a single neuron learning it uses Thermal perceptron learning rule that was introduced in section 3.2. At a global level competition between neurons is incorporated. This approach makes learning more efficient and allows for obtaining more complex structures.

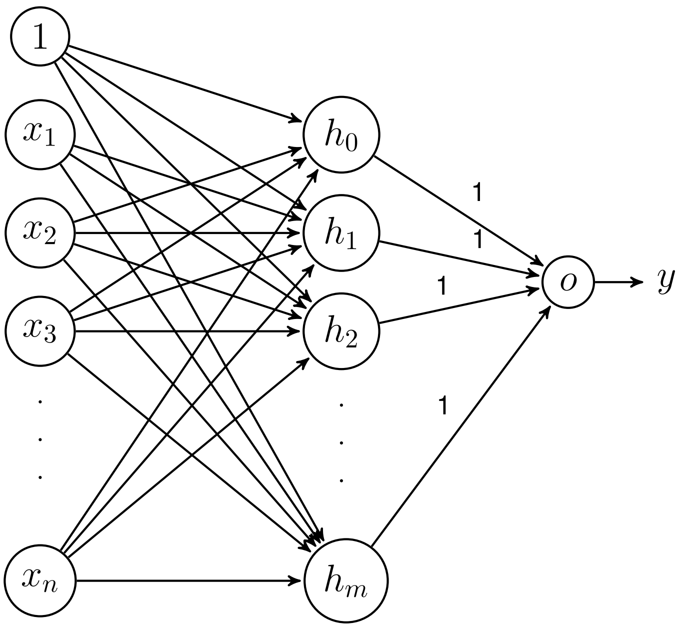


Figure 4.2: Structure of C-Mantec algorithms for two-class function

4.2 Marchand's algorithm

Algorithm has been proposed by Mostefa Golea and Mario Marchand in the article "A Growth Algorithm for Neural Network Decision Trees"[6]. Algorithm 6 describes building of a feed-forward network for a two-class classification with only one hidden layer. This structure is shown in figure 4.3.

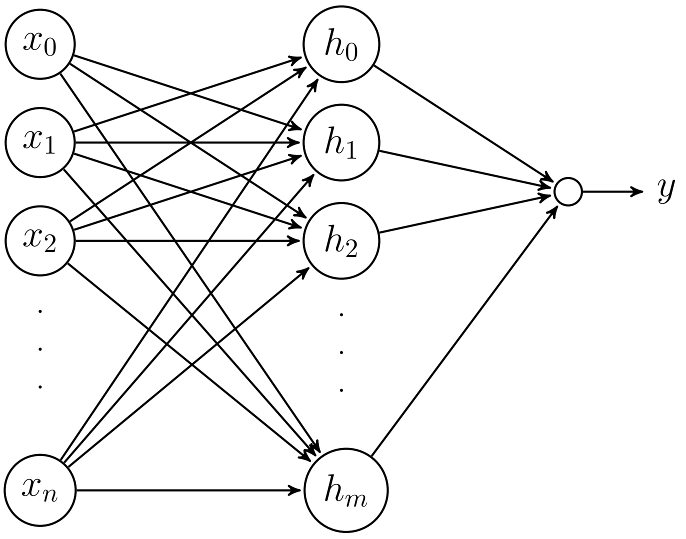


Figure 4.3: Network built with Marchand's algorithm

The algorithm describes how to add neurons into a hidden layer one by one – when the neuron helps ANN to classify at least one more new example to the appropriate class. The algorithm ensures that a new neuron does not break correctly-classified examples by neurons added previously. Algorithm ensures this by setting weights between newly-added unit and output unit according to:

$$w_k = \begin{cases} \frac{1}{2^k} & \text{if neuron k belongs to class 1} \\ \frac{-1}{2^k} & \text{if neuron k belongs to class 2} \end{cases}$$

This algorithm works with two sets T_k^+ and T_K^- - they represent patterns that are not correctly classified in step k.

Algorithm 6 Marchand's algorithm

```
procedure MarchandAlgorithm
    $k = 0$
    $T_0^+$ = set of samples from class 1
    $T_0^-$ = set of samples from class 2
    while $T_k^+ \neq \{\}$ & $T_k^- \neq \{\}$ do
        $k = k + 1$
        $w = 0$
        Create neuron $n_k$ in hidden layer that satisfies one of next cases.
        case output of neuron $n_k$ is 0 for all samples from class 1 and 1 for at least one case from class 1 then:
            $T_{k+1}^- = T_k^-$
            $T_{k+1}^+ = \{t \mid t \in T_k^+$ & output of neuron $n_k$ for $t \neq 0\}$
            $w = \frac{1}{2^k}$
        end case
        case output of neuron $n_k$ is 0 for all samples from class 1 and 1 for at least one case from class 2 then:
            $T_{k+1}^+ = T_k^+$
            $T_{k+1}^- = \{t \mid t \in T_k^-$ & output of neuron $n_k$ for $t \neq 0\}$
            $w = -\frac{1}{2^k}$
        end case
        Add neuron to hidden layer with weight $w$
    end while
end procedure
```

4.3 New Constructive Algorithm

New Constructive Algorithm (**NCA**) was proposed in 2009[9]. NCA creates a unique topology shown in figure 4.4. Each hidden layer receives outputs of each preceding layer (an input layer and hidden layers). Whereas the output layer receives all hidden layer outputs. Every neuron from hidden layers uses a sigmoid activation function.

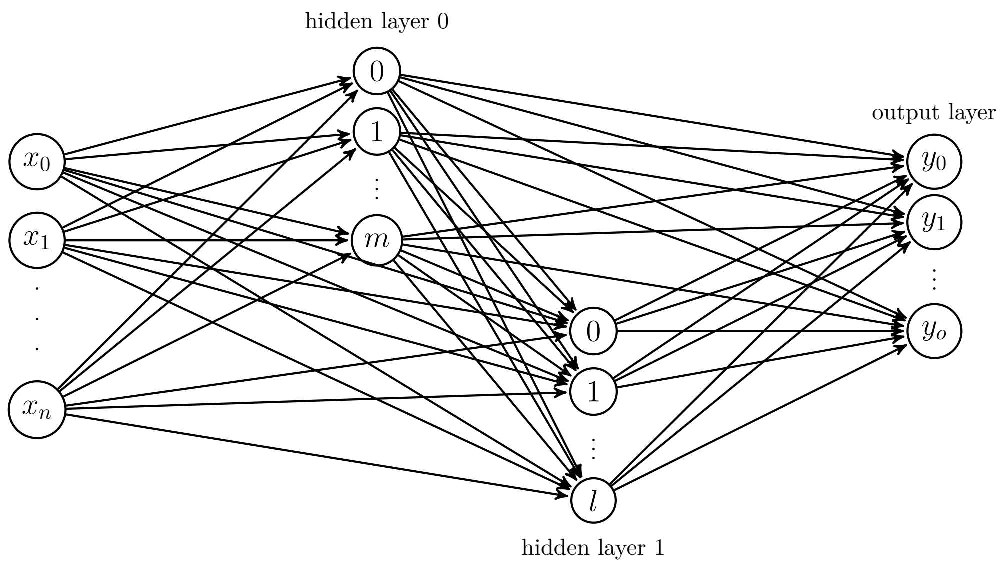


Figure 4.4: Structure of NCA

In figure 4.5 we can see major steps of NCA. These steps are going to be described in detail further on.

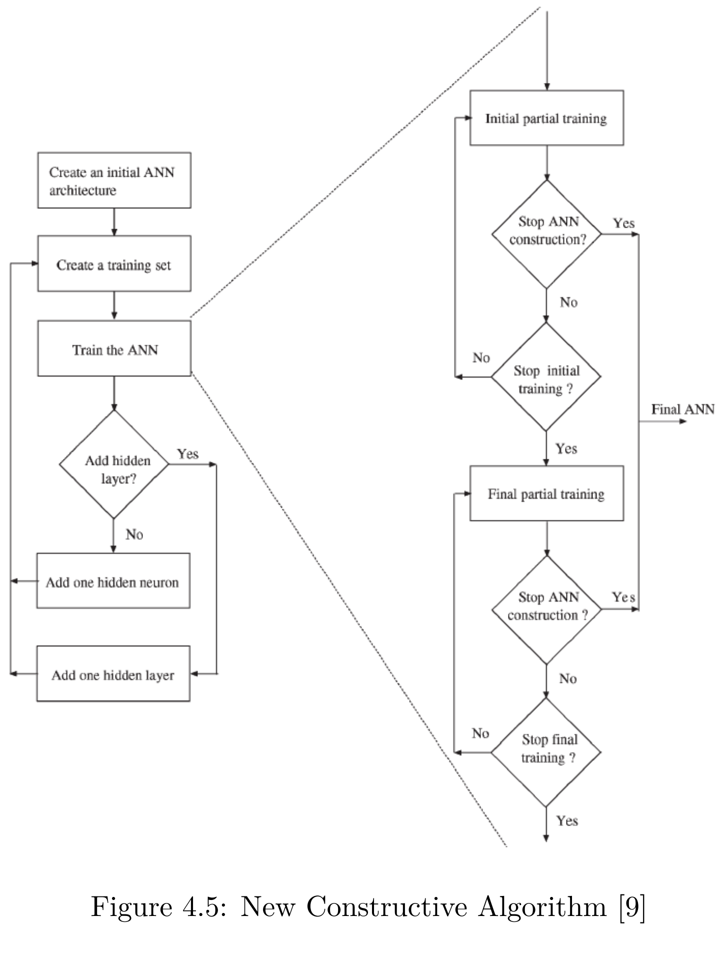


Figure 4.5: New Constructive Algorithm [9]

- **Create an initial ANN structure**
  Choose ANN with three layers - input layer, one hidden layer and one output layer. Hidden layer contains one neuron.
  Randomize all weights in ANN within a small range. Label the hidden layer C and its neuron I.

- **Create a training set**
  Create a training set with **AdaBoost** for the neuron I from layer C. Training set for the first neuron I and first layer C is the original training set.

- **Initial partial training**
  Use **backpropagation** to train neuron I from layer C using the set created in the previous step.

- **Stop ANN construction?**
  Check termination criterion for stopping ANN construction and return created network if criterion is fulfilled.

- **Stop initial training?**
  Compute error E of ANN on training set. If an error is reduced by a predefined value, go to the step **Initial partial training**, otherwise continue.

- **Final partial training**
  Add a small amount of noise to all input and output connection weights of neuron I. Usually Gaussian distribution with a mean of zero and a variance of one is used. Train neuron I using **backpropagation**.

- **Stop final training?**
  Compute error E of ANN on training set. If an error is reduced by a predefined value, go to step **Final partial training**, otherwise remove label I and continue.

- **Add hidden layer?**
  Check the criterion for adding a new hidden layer. If criterion is fulfilled go to **Add one hidden layer**. Otherwise go to **Add one hidden neuron**.

- **Add one hidden neuron**
  Add new neuron to layer C and label it I. Initialize its input and output connections with zero and go to the step **Create a training set**.

- **Add one hidden layer**
  Add a new hidden layer with one neuron above layer C. Label this layer C and the first neuron I and randomize all weights in ANN around zero. Continue with the step **Create a training set**.

4.4 Cascade-Correlation

Algorithm **Cascade-Correlation** was proposed in 1990 [4]. This algorithm combines two ideas. The first is a cascade structure, where neurons are added one by one, and after an addition they never change again. The second one is a learning algorithm which creates new hidden neurons. For every neuron, algorithm maximizes the magnitude of the **correlation** between new unit's output and residual error signal.

Algorithm starts with inputs and outputs but no hidden units. The number of inputs and outputs is determined by a problem definition. Every output is connected to all inputs. Either any linear function or any non-linear function can be used as an activation function. When a new hidden neuron is added, it receives connections from all inputs and all previously added hidden neurons - thereof the name "cascade". Such a structure is illustrated in figure 4.6

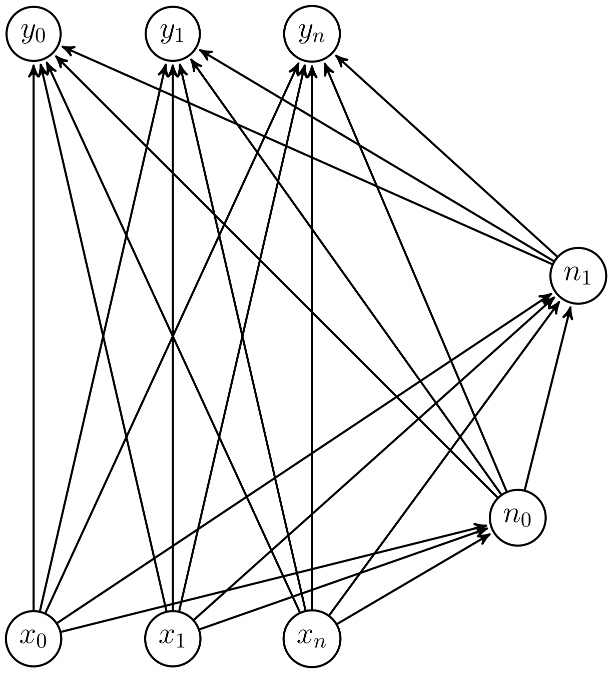


Figure 4.6: Structure of Cascade-Correlation algorithm

Algorithm defines **the sum of magnitude of correlation** over all output units o as:

$$S = \sum_{o} \left(\sum_{p} \left(V_p - \overline{V}\right)\left(E_{p,o} - \overline{E_o}\right) \right)$$

Where $E_{p,o}$ is error observed at unit o with pattern p, V_p is value of candidate unit. The quantities $\overline{E_o}$ and $\overline{V}$ are values averaged over all patterns. The task is to maximize S. For this task very similar derivation to backpropagation one is defined in the original paper:

$$\frac{\partial S}{\partial w_i} = \sum_{p,o} \sigma_o \left(E_{p,o} - \overline{E_o}\right) f'_p I_{i,p}$$

Where σ_o is the sign of correlation between candidate's value and output o, f'_p is the derivative for pattern p of candidate unit's activation function and $I_{i,p}$ is the input of the candidate for input unit i and pattern p. After computing $\frac{\partial S}{\partial w_i}$, gradient ascent can be performed. This step trains only output units and the usage of backpropagation or quickpropagation is recommended.

Further algorithm 7 is provided. The provided algorithm uses equations described above.

Algorithm 7 Cascade-Correlation

```
 1: procedure Cascade-Correlation
 2:     $minimalErrorStep$ = user defined minimal error step
 3:     $maxIterations$ = user defined maximal number if iterations
 4:     $error$ = error treshold defined by user
 5:
 6:     $network$ = network with fully connected input/output layer and zero hidden neurons
 7:     $iterations = 0$
 8:     $error = TrainOutputUnits(network)$
 9:     while $iterations < maxIterationsanderror < errorTreshold$ do
10:         $candidates = generateCandidates(network)$
11:         $bestCandidate = trainCandidates(network, candidates)$
12:         $network.addHiddenNeuron(bestCandidate)$
13:
14:         $error = TrainOutputUnits(network)$
15:         $iterations + +$
16:     end while
17: end procedure
```

4.5 Cascade 2 Algorithm

The algorithm **Cascade 2** was proposed and implemented by Scott E. Fahlman. He also wrote an article[3] in 1996, but the article was never published and it is not possible to get to the original article anymore. Thankfully, it is possible to get C port of the original implementation[1]. Currently, it is possible to find another implementation of this algorithm in C based neural network library **FANN**. Work done on **FANN library** is well documented in the work "Large Scale Reinforcement Learning using Q-SARSA(λ) and Cascading Neural Networks"[14].

The Cascade 2 is modified Cascade-Correlation algorithm, described in previous section. Algorithm changes the way, it trains candidates and adds them to network. The main difference is, that this algorithm also trains output weights for candidates. Algorithm is trying to minimize the difference between the error of the output layer and the input from candidate. Difference between candidate and output can be computed as follows:

$$S2 = \sum_{o} \left(\sum_{p} e_{p,o} - o_p \cdot w_o \right)$$

$e_{p,o}$ is the error observer at output o for pattern p. This error can be computed as $e_{p,o} = y_{p,o} - y'_p$, where y is the desired output and y' is output computed by networks. o_p is output of candidate for pattern p and w_o is weight between candidate unit and output o.

To minimize $S2$, $\frac{\partial S2}{\partial w_o}$ is calculated.

$$\frac{\partial S2}{\partial w_o} = -2 \sum_{o} (o_p \cdot w_o - e_{p,o}) \cdot o_p$$

After computing of $\frac{\partial S2}{\partial w_o}$, gradient ascent can be performed.

[1]The C port of original Lisp code is possible to find at https://github.com/gtomar/cascade or http://www.cs.cmu.edu/~sef/sefSoft.htm

4.6 Percentage Average Synaptic Activity

Pruning algorithm was proposed in the paper Dynamic Pruning In Artificial Neural Networks [1] by E. R. Caianiello , G. Orlandi , F. Piazza , A. Uncini , E. Guminari and A. Ascone. In that paper, new algorithm for eliminating units in a way, that performance of network did not worsen over the time. This algorithm develops a multilayer perceptron. To verify if connection is necessary or can be removed, a simple formula was defined. For understanding of this formula, steps to derive this equation are shown.

First, we need to define *synaptic activity* of connection from neuron i to neuron j:

$$a_{ij}(p) = (w_{ij} \cdot Act_j(p))^2$$

This equation is defined relatively to training pattern p. In this equation $Act_j(p)$ is the activation of neuron j for the input pattern p.

This value can be averaged through the whole training as follows:

$$\hat{a}_{ij} = \frac{\sum_{p}^{N_p} a_{ij}(p)}{|N_p|}$$

Where N_p is the whole training set and $|N_p|$ is its size.

As we defined average synaptic activity between neurons i and j, we need to define *synaptic activity* of neuron, which is defined as:

$$A_i = \sum_{i}^{N_i} \hat{a}_{ij}$$

where N_i are all input neurons of neuron i. Now, we can define the **Percentage Average Synaptic Activity**:

$$PASA_{ij} = \frac{100 \cdot \hat{a}_{ij}}{A_i}$$

As we can see, **PASA** is defined as average activity of connection relatively to neuron's average activity. As we defined **PASA**, we still need to define the function, we can compare this value to. Lets define threshold function $Th(ep)$ where ep is number of epoch in learning. Parametric threshold function allows us to change threshold dynamically with learning epochs. The original paper come with two possible comparisons of **PASA** and $Th(ep)$ [4.1,4.2].

$$\begin{cases} PASA_{ij} < Th(ep) & \text{connection } i - j \text{ is pruned} \\ otherwise & \text{connection } i - j \text{ is retained} \end{cases} \tag{4.1}$$

$$\begin{cases} \frac{\frac{100}{|N_i|} - PASA_{ij}}{\frac{100}{|N_i|}} < Th(ep) & \text{connection } i - j \text{ is pruned} \\ otherwise & \text{connection } i - j \text{ is retained} \end{cases} \tag{4.2}$$

First equation is affected by the number of synapses to neuron j. With higher number of inputs, **PASA** is lower. This can be modified by introducing $\frac{100}{|N_i|}$ to equation. In the original paper the function $Th(ep)$ was defined as follows:

$$Th(ep) = ve^{-\frac{1}{2}(\frac{ep-m}{\sigma})^2}$$

where v,m and σ are constant values. This function is known as *Gaussian function.*

4.7 Neuroevolution

Neuroevolution is a bit special group within constructive algorithms. These algorithms use other **Soft-computing** methods to develop topology and weights. Some of these methods develop only topology, whereas others are able solve both tasks.

We can put algorithms into two groups - algorithms that use **direct encoding** and algorithms that use **indirect encoding**. When **direct encoding** is used, concrete neurons and connections are represented by genes. On the other hand **indirect encoding** describes how to construct neural network. It allows to create compact genomes that are able to create bigger neural networks or are able to define simply multiple occurrences of the same sub-network.

Many algorithms, that fall within this group have been developed. In the next section two of them are described.

4.7.1 Cellular Encoding

Cellular Encoding is a method for encoding neural networks. This technique developed in 1994 [7] uses **indirect encoding**. Algorithm introduces cells that execute "cellular code" and turn into neurons when they finish code execution. The code is represented by a tree (also called "cellular code"). This tree consists of instructions and its nodes – number of nodes depends on instruction. Roots with terminal instructions doesn't contain nodes, while nonterminal ones contain one or two – depending on whether instruction divides into two cells or not. Every cell executes instructions according to its position in the code. When the cell gets to the end of code, it "dies" and turns into neuron.

Algorithm uses the initial tree. There are two types of trees defined – cyclic and acyclic as shown in figure 4.7. This tree is used for initial development.

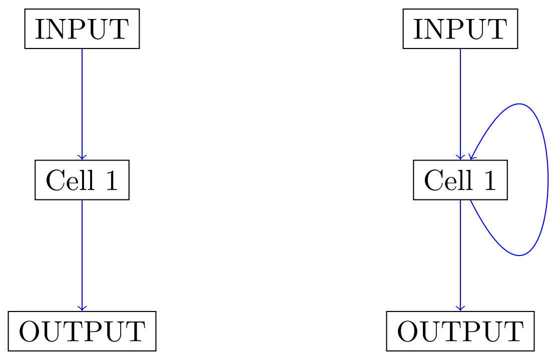


Figure 4.7: Initial graphs of cells. There is an acyclic graph on the left side and a cyclic graph on the right side.

Further, the algorithm is run for an inspiration with a simple "cellular code". Algorithm starts with a single cell pointing to the beginning of the code as shown in figure 4.8.

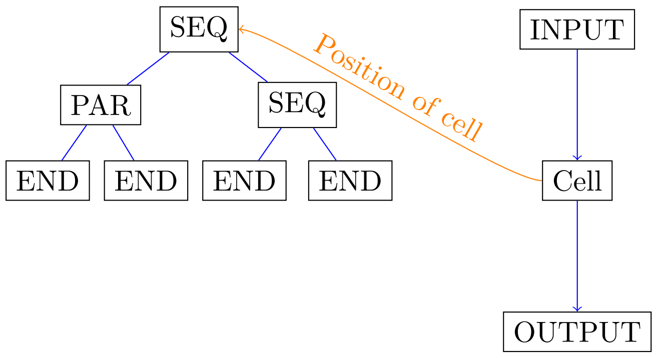


Figure 4.8: Starting point of algorithm

Cell 1 then interprets the instruction – **SEQ** in this case and it results in creating a new cell with an input connection to the first cell and its output to the output as shown in figure 4.9

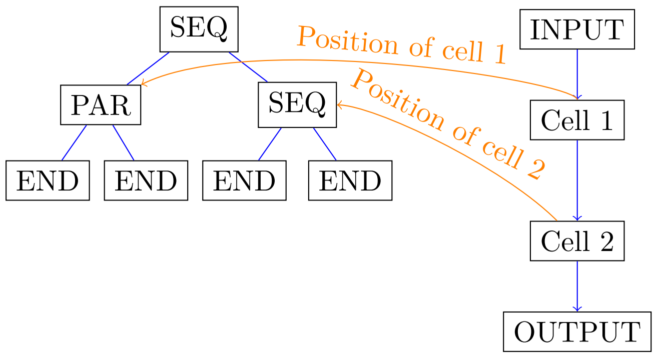


Figure 4.9: State of algorithm after first step

In the next step of algorithm, all cells interpret next instructions again. Cell 2 in this case points to **SEQ** and cell 1 to **PAR**. Instruction **PAR** is also division one. This instruction creates a new cell, that inherits the same inputs and outputs as the first cell – that means a parallel division. This can be seen in figure 4.10.

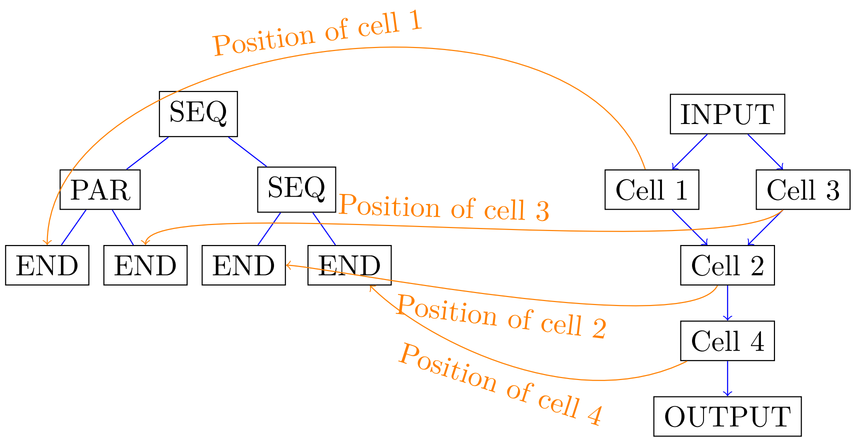


Figure 4.10: Developing of cells

In the last step all the cells point to the instruction **END**. This instruction turns cells into neurons. This conversion can be seen in figure 4.11.

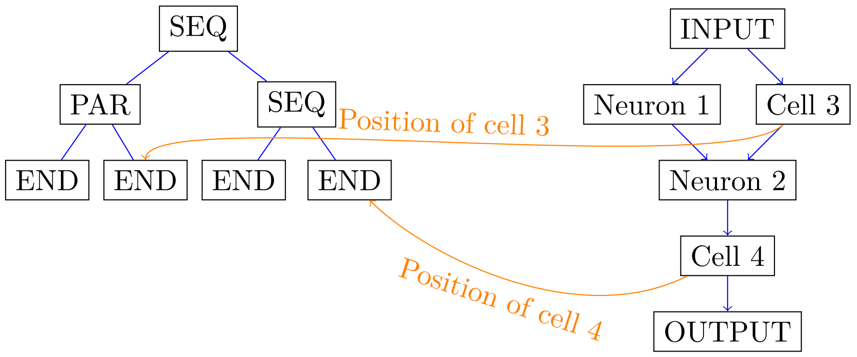


Figure 4.11: Dying of cells and creation of neurons

The original paper introduces many instructions, nevertheless the instructions provided below are satisfactory for development of every possible topology. Other instructions are just making process of development quicker and cellular code more compact.

- **SEQ** – *sequential division* of a cell into two. The first created cell inherits all input links and second outputs. These two cells are connected with weight 1.
- **PAR** – *parallel division* is the second type of a division. Both new created cells inherit inputs and outputs from the original cell.
- **END** – *ending-program symbol* ends editing of a cell.
- **DECBIAS/INCBIAS** - these symbols modify bias of neuron – increase or decrease it.
- **DECLR/INCLR** – these symbols modify a value of link register
- **VAL-/VAL+** – these symbols set a value of link register to -1 or +1
- **CUT** – it modifies topology by removing link pointed by link register.

4.7.2 NEAT

Algorithm **NeuroEvolution of Augmenting Topologies** was invented by Stanley and Miikkulainen [19]. This method develops both topology and weights. This method is based on **genetic algorithms** – it uses direct encoding of genes and crossover.

This algorithm develops linear genome, consisting of two types of genes (*Connection Gene* and *Node Gene*). The ability of developing topology comes with problems – how to make crossover of two different genomes with different sizes and how to crossover two same networks with different topology. **NEAT** comes with solution – every genome gets its *ID*, that is never changed. This *ID* is inherited, so it is possible to get ancestor of concrete gene. This solves the problem of making crossover of two different genomes. When crossover is done, all genes are paired with same *ID*s and one gene of group gets to newly created genome.

When a new topology is created – i.e. a new neuron or connection is added, it can break fitness of newly created *individuals*. To avoid losing such an individual – its topology can be better, but weights are not developed, **NEAT** comes with splitting of individuals into *species*. This allows to create continuous development of genomes with different topologies. Every time, a new genome that is different from other *species* is evolved, a new species is created. This dissimilarity is computed from number of unpaired genes (number of not paired IDs) and the difference in weights.

$$\delta = \frac{c_1 E}{N} + \frac{c_1 D}{N} + c_3 \cdot \overline{W} \tag{4.3}$$

Equations 4.3 describes dissimilarity of two genomes. c_1, c_2, c_3 are coefficients affecting impact of factors. N is length of longer genome, $\overline{W}$ is average of differences between weights. E is number of **excess genes** – the number of genes of one individual, that have higher ID than the highest ID of gene from the second genome. D is the number of genes, that can not be paired – ID of genes are in only one genome.

Chapter 5

Genetic Programming

Genetic programming falls into the category of evolutionary algorithms. This algorithm works either with linear code or code encoded into trees. In this chapter, genetic programming for code encoded into trees is introduced and methods for selection, mutation and crossover are described.

5.1 Population initialization

Initialization of population is the first step of genetic programming algorithm. The way population is initialized affects the speed of convergence to local (global) optimum. Usually individuals with short code decrease diversity whereas too complex and deep code tree can result in inefficient solutions. For initialization of trees, usually two methods are used – **grow method** and **full method**. These methods need one parameter – the “maximum tree depth”. This parameter restricts the size of tree. Apart from “maximum tree depth” it receives also set of terminals and nonterminals, it generates tree from.

The **Grow** method 8 creates tree, where in every step where depth is smaller than maximum depth, node is selected from all symbols. In the maximum depth, node is selected only from terminal symbols [22].

Algorithm 8 Grow method

```
procedure GROW(depth)
    if $depth < maximumDepth$ then
        $node = random(Terminal \cup Non - -Terminal)$
        for $i : node.children$ do
            $node.child_i = grow(depth + 1)$
        end for
    else
        $node = random(Terminal)$
    end if
    return node
end procedure
```

The **Full** method 9 is similar to **Grow** method, but generates trees of the depth equal to maximum depth – it selects only non-terminal symbols when the depth is smaller than maximum depth.

Algorithm 9 Full method

```
 1: procedure FULL(depth)
 2:     if $depth < maximumDepth$ then
 3:         $node = random(Non - -Terminal)$
 4:         for $i : node.children$ do
 5:             $node.child_i = grow(depth + 1)$
 6:         end for
 7:     else
 8:         $node = random(Terminal)$
 9:     end if
10:     return node
11: end procedure
```

There exists one more method – "ramped half-and-half". This method combines the two previous. It initializes half of population by **grow method** and the second half by **full method**.

5.2 Operators

Operators are crucial part of genetic programing. Operators define the way individuals mutate or crossover. During experiments, new operators were implemented to achieve better results. We divide operators into two categories – mutations and crossover.

5.2.1 Mutations

Mutation is unary operator and it changes genetic information of individual. This operator is useful, when generation is approaching optimum and crossover between individuals makes big differences in code trees [16].

Subtree mutation 5.1 is basic mutation operator, which selects random subtree in individual and replaces it with randomly generated tree. Newly generated subtree must be generated with reasonable depth, any algorithm mentioned in Genetic code initialization can be used.

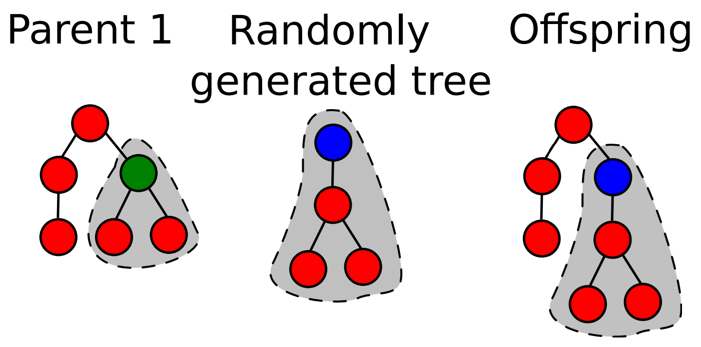


Figure 5.1: Subtree mutation

Shrink mutation 5.2 reduces the complexity of code. This operator selects random subtree that is replaced by subtree with the first instruction being terminal. This can be pictured on following equation: $pow(x) \cdot (x+0)$, that can be turned after application of this operator to $pow(x) \cdot x$.

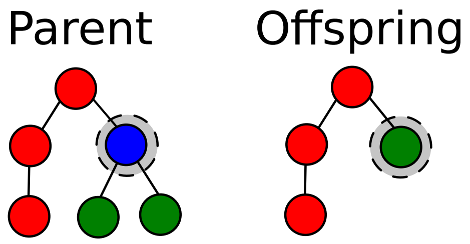


Figure 5.2: Shrink mutation

Node replacement mutation 5.3 replaces instruction by another with the same signature (arity, return or parameter types) [10].

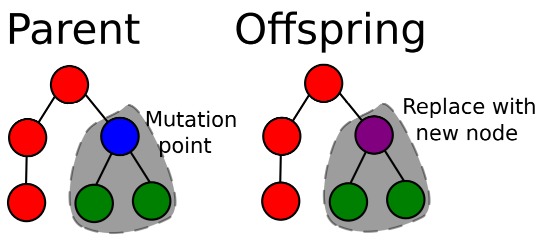


Figure 5.3: Node replacement mutation

Hoist mutation 5.4 is very specific operator. This operator can be though as opposite to shrink mutation and is also useful for reducing complexity of code. It selects random subtree and replaces whole tree node with it – the selected subtree is new solution.

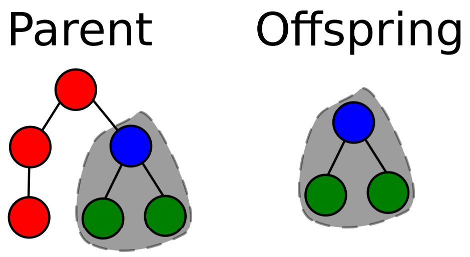


Figure 5.4: Hoist mutation

Subtree permutation 5.5 is the last described mutation operator. This operator is applicable only, when modified instruction is not commutative. Operator switches operands (subtrees of node) of instruction. This can be seen on formula change from a/b to b/a.

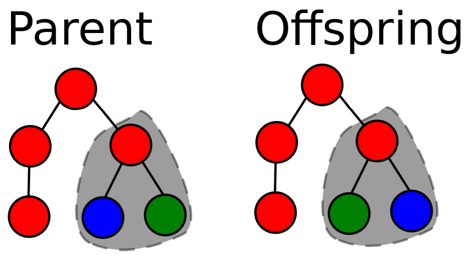


Figure 5.5: Subtree permutation

5.2.2 Crossover

Crossover operators are binary operators. These operators select two individuals from generation and creates new one using their genetic information.

Subtree crossover [17] is the first introduced operator. This operator selects node in both parents and switches these two subtrees as can be seen in figure 5.6.

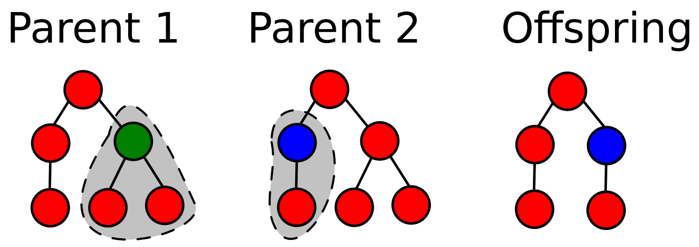


Figure 5.6: Subtree crossover

Arity-2 combination is specific operator. This operator combines both parents with usage of binary instruction. This combination can be seen in figure 5.7.

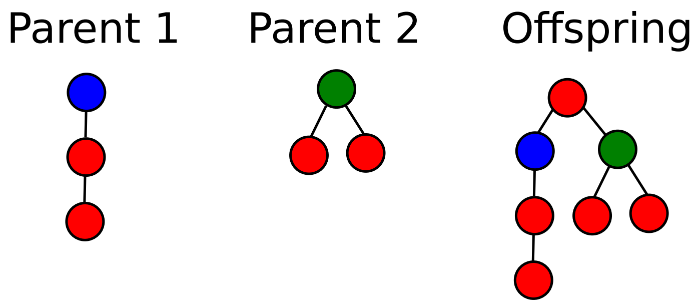


Figure 5.7: Arity-2 combination

Chapter 6

Implementation

This chapter describes requirements and implementation of neccessary parts for the thesis. Description of specific modules and components is also provided.

6.1 Requirements for implementation

The essential requirement for the final design was the possibility of extension of implemented features for the future. Due to this fact the code is divided into 3 logical parts. The first part is library for neural networks. The second part is library for evolutionary algorithms and the last part consists of experiments incorporated in the thesis. Implementation of libraries is provided in the two following sections. The second requirement was speed and portability. For this fact the language C++ was chosen.

6.2 Neural network library

For the specific needs of this thesis, a personal library was developed. This library is called **NeuralNetworkLib**. This unit is not completely original, it was originally developed as a part of my master's degree studies subject. This library is provided as an open-source[1]. For this thesis, this library has been extended by:

- Recurrent network
- Cellular encoding
- Cascade network implementation
- Constructive algorithms
- **AVX** implementation of basis function
- Problem sets
- Serialization of networks

[1]Library can be found at https://github.com/Shin-nn/NeuralNetworkLib or http://gitlab.ishin.cz/shin/NeuralNetworkLib

This library aims to provide high-speed functionality framework as well as simple object-oriented design. Due to the fact, that these objectives go against each other, I dropped this clean object-oriented design. Every element of this library knows how to serialize itself as "JSON object" as well as deserialize. Format JSON was chosen for its lightness and easy readability. The ability and uniformity of serialization of every object comes with the ability to simply combine parts together and store networks to disks and load once again. We can imagine this functionality on serialized neuron in listing 6.1.

```
{
"class" : "NeuralNetwork::Neuron",
"activationFunction" : {
        "class" : "NeuralNetwork::ActivationFunction::Sigmoid",
        "lambda" : -0.800000
},
"basisFunction" : {
        "class" : "NeuralNetwork::BasisFunction::Linear"
},
"id" : 3,
"output" : 0.483658,
"value" : -0.081738,
"weights" : [-8.339814, 3.148232, 3.169152, 6.976029]
}
```

Listing 6.1: Neuron serialised as JSON

This library is divided into logical parts, each representing part of neural network. Further, each part is introduced.

6.2.1 Activation functions

Activation functions are first essential part of this library. These classes represent different activation functions. We can see implemented functions and class dependency in figure 6.1. Each function provides function to compute output of neuron from its value – function **operator()** as well as function for computing derivation – function **derivatedOutput**. Derivated output is used for example during backpropagation.

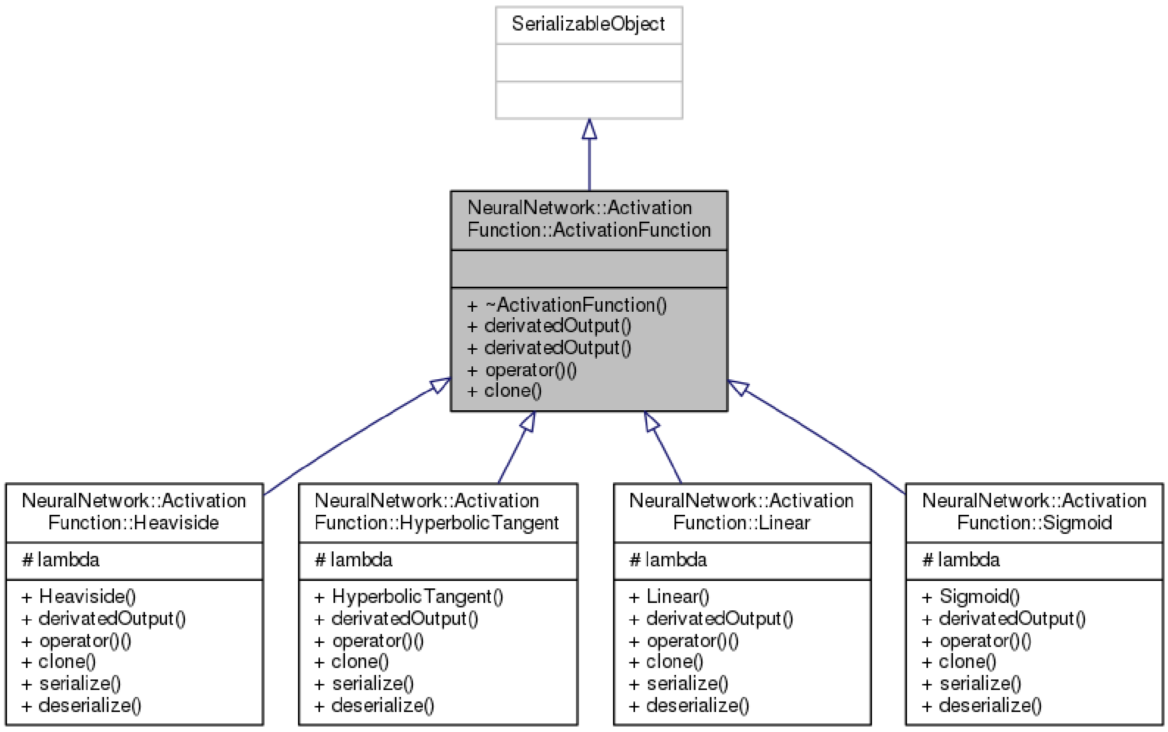


Figure 6.1: Activation function diagram

6.2.2 Basis functions

The second part of library are basis functions. Each function implements the computation from weights and inputs – function **operator()**.Due to the fact, that in most cases linear basis function is used, it was tuned to be as efficient as possible.

To satisfy this goal, linear basis function was implemented with streaming instructions – **AVX** and **SSE**. In the time of compilation, compiler tool decides, whether instruction set is available on its platform. This allows to write portable program as well as target high speed. The usage of these instruction sets allows neural network to run multiple times quicker. Given the fact, that more than 95 percent of time spent on evaluating neural network is in linear functions, the speed up is in the case of **SSE** instruction set almost by 4 and in the instruction set of **AVX** almost by 8.

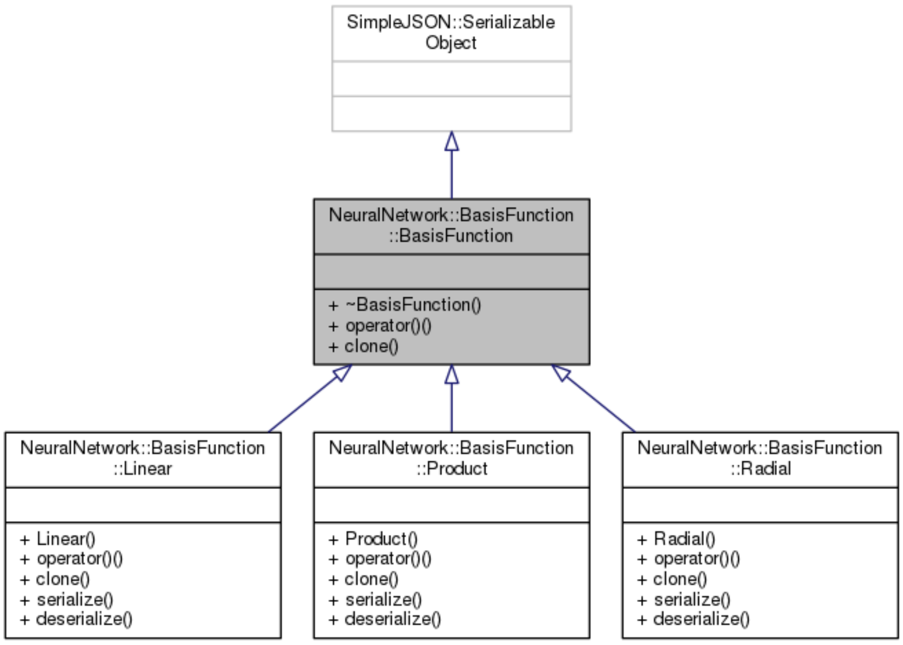


Figure 6.2: Basis function diagram

6.2.3 Neuron

Neuron is the most important point of this library. Its interface and attributes can be seen in figure 6.3. The neuron's main, and the most computationally intensive function is **operator()**. This function receives vector of inputs, uses basis function to compute inner value of neuron and activation function to compute its output, which is returned. This two values, are then stored inside the neuron for further usage (backpropagation, etc.). This composition allows neuron to be used in all types of discrete networks. Networks' responsibility is to pass inputs to neuron in correct order, and the rest is done inside neuron. Further, the neuron provides interface for changing and getting weights, activation and basis function. It also allows to resize number of inputs, to be able to change topology on the run.

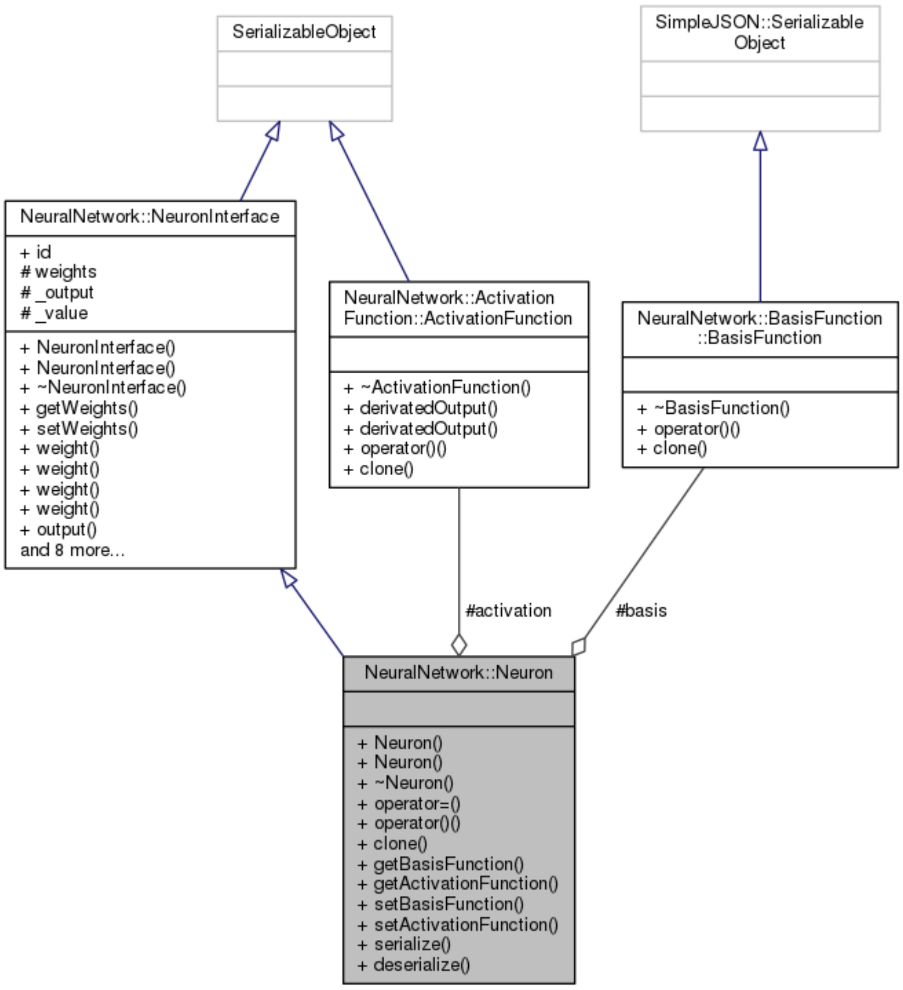


Figure 6.3: Neuron diagram

6.2.4 Learning algorithms

Another part of library are learning algorithms. Implemented algorithms relevant to this thesis are backpropagation and quickpropagation. Algortihms can be found in namespace **LearningAlgorithms**. These two algorithms can be used for learning any feed-forward network and are used for learning output layers in algorithms Cascade-Correlation and Cascade 2.

6.3 Evolutionary algorithms library

This library serves as multipurpose library. Library is provided as open-source[2]. It unifies algorithm such as genetic programing, genetic algorithms and so on. Library is based on common base – template called **EvolutionaryAlgorithm**. This template defines the process of initializing new generation and generating new one. It provides all core functionality to specific routines – only operators and types of individuals are necessary to specify. The workflow can be modified in many ways - number of selected individuals from previous

[2]Library can be found at http://gitlab.ishin.cz/shin/EvolutionaryAlgorithms

generation, size of generation and terminal criterion. It is also responsible for creating statistics of generations.

6.3.1 Selection Operators

Library implements several selection operators. These operators can be used for selecting individuals to next generation or individuals for mutation or crossover. Implemented operators and corresponding diagram can be seen in figure 6.4. Operators are implemented as templates. Further, implemented operators are described.

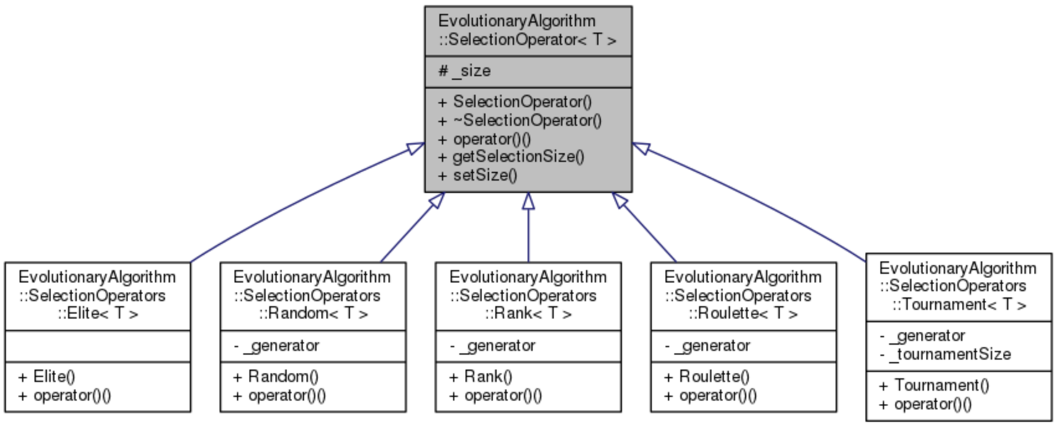

EvolutionaryAlgorithm ::SelectionOperator< T >
# _size
+ SelectionOperator()
+ ~SelectionOperator()
+ operator()()
+ getSelectionSize()
+ setSize()
EvolutionaryAlgorithm ::SelectionOperators ::Elite< T >
+ Elite()
+ operator()()
EvolutionaryAlgorithm ::SelectionOperators ::Random< T >
- _generator
+ Random()
+ operator()()
EvolutionaryAlgorithm ::SelectionOperators ::Rank< T >
- _generator
+ Rank()
+ operator()()
EvolutionaryAlgorithm ::SelectionOperators ::Roulette< T >
- _generator
+ Roulette()
+ operator()()
EvolutionaryAlgorithm ::SelectionOperators ::Tournament< T >
- _generator
- _tournamentSize
+ Tournament()
+ operator()()

Figure 6.4: Selection operators diagram

Elite selection selects first best N individuals. This operator is usually used, when best individuals should be placed to new generation.

Roulette Wheel selection selects individuals according to their fitness. Selection can be imagined on a roulette wheel, where every place represents individual and the proportion of size is the same as proportion of fitness. This operator suffers from problems, where fitness differs very much – that way, only few best individuals are selected.

Rank Selection is trying to reduce the problems, that comes with Roulette Wheel selection. It works, the same way as previous operator, but the proportion of space is computed by their rank in population. Worst individual gets rank 1, second worst gets rank 2 and the best gets rank N – size of population. There is one problem with this operator – as diversity of selected individuals increases, the convergence can be slower.

Tournament selection comes with an idea of comparing random individuals to each other. Operator selects randomly k (the size of tournament) individuals from the population. The individual with best fitness is the winner of tournament.

6.3.2 Genetic programing

Genetic programing is based on previously described class **EvolutionaryAlgorithm**. It provides individuals with trees. Every tree can contain zero, one or two nodes. Number of nodes depends on concrete instruction. Every instruction must specify its number of nodes, it uses.

6.3.3 Population initialization

Library implements all initialization methods described in section 5.1. These methods can be found in namespace **GeneticPrograming::InitializationFunction** as we can see in figure 6.5. These methods implement method **operator()()** that returns initialized tree.

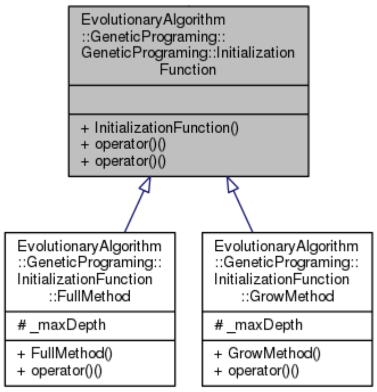

EvolutionaryAlgorithm ::GeneticPrograming:: GeneticPrograming::Initialization Function
+ InitializationFunction()
+ operator()()
+ operator()()
EvolutionaryAlgorithm ::GeneticPrograming:: InitializationFunction ::FullMethod
# _maxDepth
+ FullMethod()
+ operator()()
EvolutionaryAlgorithm ::GeneticPrograming:: InitializationFunction ::GrowMethod
# _maxDepth
+ GrowMethod()
+ operator()()

Figure 6.5: Population initialization methods

6.4 Cascade-Correlation

This algorithm is implemented in class **ConstructiveAlgorihtms::CascadeCorrelation**. This class provides all neccessary functions for constructing cascade-network. This algorithm needs many configurations, thus functions for setting or getting following configurations are provided:

- **Number of candidates** – number of candidates generated in each epoch
- **Maximum candidate learning iterations** – maximal number of iterations, candidates are learned
- **Error treshold** – when error on learning set is lower than threshold, algorithm ends construction
- **Radom weight range** – range, where weights are generated
- **Maximum output layer learning iterations** – maximal number of iterations for learning output layer
- **Maximum output layer learning iterations without change** – maximal number of iterations for learning output layer, when response is not getting better
- **Maximum candidate learning iterations without change** – maximal number of iterations for learning candidates, when correlation is not increasing

Entrance function for construction of network is **construct(vector of TrainingPatterns)**. Function creates Cascade network with empty hidden layer. Then it adds new units according to algorithm described in section 4.4. Given, the fact, that Cascade-Correlation and Cascade2 algorithm differ only in the way, they train candidates (and their outputs), class provides virtual function – **trainCandidate()**, that is overridden by Cascade2. This function returns candidate neuron and weights to output units.

6.5 Cascade 2

Cascade 2 algorithm is implemented in class **ConstructiveAlgorithms::Cascade2** and inherits from described class **ConstructiveAlgorithms::Cascade2Correlation**. This class overrides method **trainCandidate()**. Given the fact, that no original documentation exists, it was complicated to reproduce the original algorithm. During development, original code was studied as well as the code provided in **FANN** library.

6.6 Cellular Encoding

Cellular Encoding is implemented in namespace **ConstructiveAlgorithms::CellularEncoding**. This class contains method **construct()**, which returns common recurrent neural network. Implementation consists of many classes, to summarize, it consists of:

- **Cell**
- **Cellular Encoding**
- **Instructions**

Every part is going to be described in details further.

6.6.1 Cells

Cell is implemented in class **ConstructiveAlgorithms::CellularEncoding::Cell**. Cells represent elements of system. Cells are modified by instructions, thus this class provides interface for modification of its status.

Every cell contains information about itself and neuron, it is going to be transformed to on the end of algorithm. More specifically, it contains:

- **position in tree** – pointer to current position in tree
- **bias** – bias value for neuron
- **life** – life, for recursive cellular code
- **links** – vector of links to the cell
- **link register** – index to current position in links vector
- **output flag** – whether cell results in output neuron
- **input flag** – whether cell results in input neuron

For all of these attributes functions enabling to set or get the value are provided. Link can be pictured as triplet $(status, weight, cell)$, where status is the status of neuron with values ON, OFF. Weight is representing weight connection between neurons and cell is identification of a cell.

6.6.2 Instructions

Part of Cellular encoding implementation represent instructions. All implemented instructions derive from base class. Every instruction needs to implement 3 functions – **run()**, **numberOfOperands()** and **toString()**.

The last function **toString()** is used for stringification of code. The second function **numberOfOperands()** is used for creating and changing code tree. Function returns number of nodes it uses. The first function **run()** represents entry point for cell. When cell gets on the move, it calls run on current instruction it points to. This function modifies cell, and its surrounding.

Every instruction is implemented in its own class and these classes are located in namespace **ConstructiveAlgorithms::CelularEncoding::Instruction**. Implemented instructions and classes are:

- **IncBias** – This instruction modifies cell by incrementing bias by one.
- **DecBias** – This instruction modifies cell by decrementing bias by one.
- **MulBias** – This instruction modifies cell by multiplying bias by one.
- **DivBias** – This instruction modifies cell by dividing bias by one.
- **SetBiasZero** – This instruction modifies cell by setting bias to zero.
- **SetBiasOne** – This instruction modifies cell by setting bias to one.
- **SetBiasMinusOne** – This instruction modifies cell by setting bias to minus one.
- **Par** – This instruction creates new cell from current one. New cell is connected parallely and copies all registers to new cell.
- **Seq** – Instruction creates new cell, that is connected sequentially to original and copies all registers to new cell.
- **End** – This instruction ends development of cell and turns cell to neuron.
- **Rec** – This instruction conditionally sets the code pointer to the beginning of code when the life > 1 and in the case of life equals one, it ends the same way as **END**.
- **Wait** – This instruction does nothing, it just stops cell for one step.
- **On** – This instruction modifies the state of link. It turns on the link specified by the link register.
- **Off** – This instruction modifies the state of link. It turns off the link specified by the link register.
- **Div** – This instruction modifies the value of link. It divides the value of link specified by the link register by 2.
- **Mult** – This instruction modifies the value of link. It multiplies the value of link specified by the link register by 2.
- **Inc** – This instruction modifies the value of link register by incrementing it by one.

- **Dec** – This instruction modifies the value of link register by decrementing it by one.
- **ValPlus** – This instruction modifies the value of link. It sets the value of link specified by the link register to 1.
- **ValMinus** – This instruction modifies the value of link. It sets the value of link specified by the link register to -1.
- **Declr** – This instruction modifies the value of link. It decrements the value of link specified by the link register by 1.
- **Inclr** – This instruction modifies the value of link. It increments the value of link specified by the link register by 1.

6.6.3 Cellular Encoding

Cellular Encoding is the class, that encapsulates whole algorithm to user and "converts" cellular code to network. Class provides functions to set initial graph, life and maximum number of steps. The last parameter was introduced for possibility of controlling algorithms run. In every step, algorithm goes over all cells that are alive and interprets instruction. When number of steps overcomes the maximum set by user, exception is thrown and algorithm is ended.

During development, implementation was tested on sample cellular codes provided in original thesis to assure correct implementation and the same results. Despite this, it was hard to obtain results that are described in original work. This is due to the fact that many relevant information are not mentioned in original work. These aspects are covered in the following paragraphs.

Timing of cells step is very important during cell development. Different timing results in different topology. To provide same results as original work, new cell must be added to execution just right after the cell it was separated from.

Cell registers must be set in divided cells during development.

Number of inputs and outputs is given by the cellular code. It is not possible to set the number of inputs or outputs and developed network must be tested on the size of input and output layers.

Terminality of symbols is not given by instruction. In the instruction set, there is only one instruction that is always terminal – **END**. Also division instructions are never terminal. Other instructions can be both terminal and non-terminal. This depends on current set specification. The ability to change the terminality of symbol allows to create specific instruction set for problems and increase convergence.

Chapter 7

Proposal of improvement

This chapter proposes two possible improvements for **Cascade-Correlation** and **Cascade 2**. In the course of experiments, I have observed that in some situations adding candidate unit to network made response of network worse. To solve this effect, two modifications are proposed. These proposals are described in the two next subsections and then in the next chapter these two proposals are evaluated.

7.1 Random search of output weights

This modification is based on the idea of wrongly generated weights between input and hidden units and output units. There are cases, when backpropagation doesn't converge, or convergence is slow. To suppress this effect, I suggest the following addition to the basic algorithm:

Instead of learning output layer by backpropagation, generate pool of $N + 1$ (the one is the original network) copies of current network. Leave one network as it is and randomize the connections between input and hidden units and output units in other networks. Then learn the whole pool. In the end of the step select network with the best response.

During experiments with this modification I observed that the profit from this modification comes with more then 95% probability in the first epochs. Thus, the number of generated networks can be modified in the following manner without reduction of profit.

$$N = \begin{cases} N' & \text{if } epoch \bmod S = 0 \\ N'/epoch & \text{otherwise} \end{cases}$$

Where S is user specified constant, in this thesis $S = 15$. N' is the maximal number of generated networks. This approach speeds the learning dramatically.

7.2 Pruning of added neurons

This modifications introduces a new variable ϵ – floating threshold for minimal convergence between added neurons. The modification can be specified as follows: When the response of network after learning of output layer is not better by the factor of ϵ, remove the last added neuron. This can be written as the following condition:

$$If(\epsilon \cdot lastError < error)\ \{\text{remove last added neuron}\}$$

Variable ϵ should be set between $(0, 1\rangle$. The smaller the variable is, the stricter the pruning.

Chapter 8

Experiments

This chapter describes typical benchmark problems and results of implemented algorithms and their improvements. These algorithms are compared to each other and to backpropagation.

As experiments, XOR, Parity of three and four inputs and Chess board 3x3 problems had been chosen.

For each experiment precise parameters and configurations of algorithms are provided for the possibility of future reproduction of experiments and corresponding results. For cellular encoding, a table describing usage of operators, initialization and other configuration is presented.

If not specified in the text of experiment, experiment was executed for a number (usually 30) of runs, then compared by resulting mean squared error and the median solution was selected. This selection of result gives us an idea of the medium solution.

8.1 Xor

Xor is typical benchmark for neural networks. This benchmark was used to show validity of algorithms and in case of cellular encoding, it was used to examine the influence of different selection operators and activation functions on the speed of convergence.

Generation size:	500
Non-terminals:	PAR, SEQ, INCLR, DECLR, INC, VAL-, INC, DEC, WAIT
Terminals:	END
Fitness function:	6.0- sum of differences between output and desired output - $size \cdot (size + 2)$
Target fitness:	Maximum
Terminal criterion:	generation >50 or fitness >5.4
Initialization:	ramped half-and-half(3)
Selection:	Elite(15)
Activation function:	Heaviside(0.0)
Mutation/Crossover:	SubTreeMutation(1.0), HoistMutation(0.5), Crossover(1.0), Permutation(0.5), NodeReplacementMutation(0.5)

Table 8.1: Table of genetic programming configuration for XOR

Table 8.1 represents configuration for cellular encoding. For this configuration, experiments with different activation functions were run. Results can be seen in figure 8.1. These results were obtained by selecting the best run from 20. In number, 80% of runs were successful in the case of Heaviside activation function. Example of generated code can be seen in appendix C. In the case of sigmoid function, the rate of success dropped to only about 8%. Due to this fact, further experiments only present data for Heaviside activation.

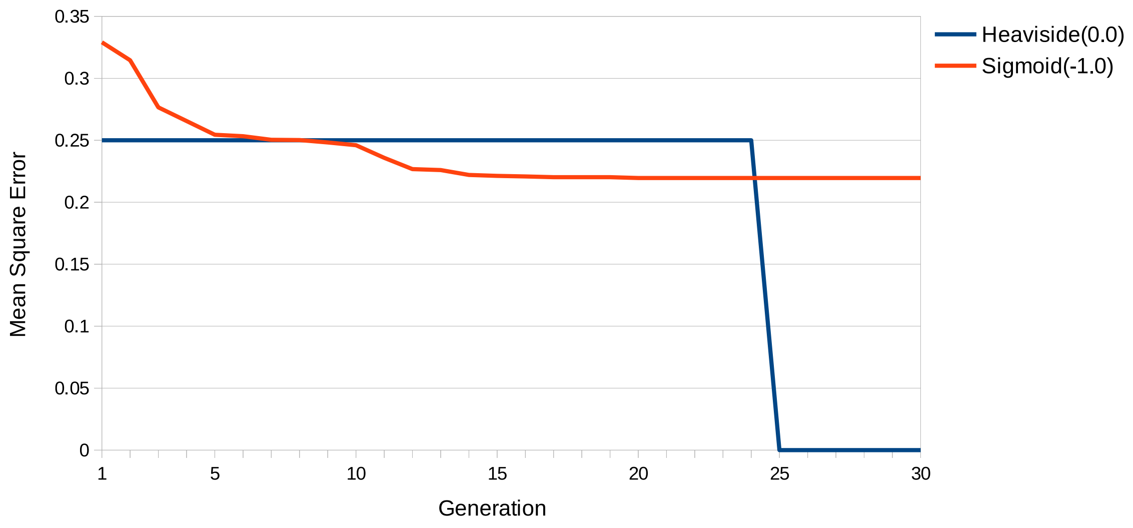


Figure 8.1: XOR cellular encoding convergence

During this experiment, different selection operators for mutations and crossover were compared from the point of convergence speed. Table 8.2 describes results of this comparison of selection operators.

Selection operator	Mean generation
Rank:	52
Roulette:	43
Tournament of size 5:	25
Tournament of size 9:	35
Tournament of size 15:	37

Table 8.2: Influence of selection operators on convergence

Cascade-Correlation and Cascade2 algorithms correctly classify all four patterns in just one step with one hidden neuron with 100% success. Due to this fact, no table with results is presented.

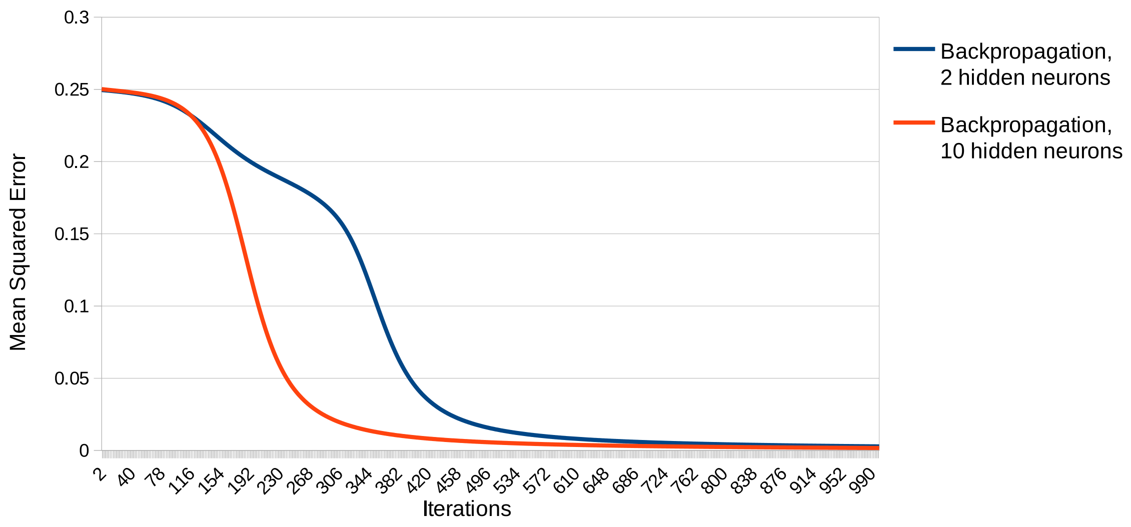


Figure 8.2: XOR backpropagation convergence

These results can be compared to backpropagation algorithm featured in figure 8.2. The result of backpropagation was selected from 20 runs of this algorithm by selecting median solution compared by mean squared error. In this example, backpropagation converges far quicker in terms of time than cellular encoding even with minimal possible topology.

8.2 Parity of 3 values

This problem represents selecting, whether the number of inputs of value 1 is odd or even.

Generation size:	500
Non-terminals:	VAL-, PAR, SEQ, WAIT
Terminals:	BIAS-, BIAS0, BIAS+
Fitness function:	10.0- sum of differences between output and desired output
Target fitness:	Maximum
Terminal criterion:	generation >50 or fitness >9.0
Initialization:	ramped half-and-half(3)
Selection:	Elite(15)
Activation function:	Heaviside(0.0)
Mutation/Crossover:	SubTreeMutation(1.0), HoistMutation(0.5), Crossover(1.0), Permutation(0.5), NodeReplacementMutation(0.5)

Table 8.3: Table of genetic programming configuration for Parity of 3 values

Table 8.3 describes configuration of cellular encoding and corresponding genetic programming. For this configuration, 30 runs of cellular encoding was executed. From these runs only 3 runs were successful which makes algorithm successful only in 10% of cases. Again, the medial result is presented in figure 8.3. This figure shows, that initially, 3 patterns are classified well, and in the future generations, the number of wrongly classified patterns decreases one by one.

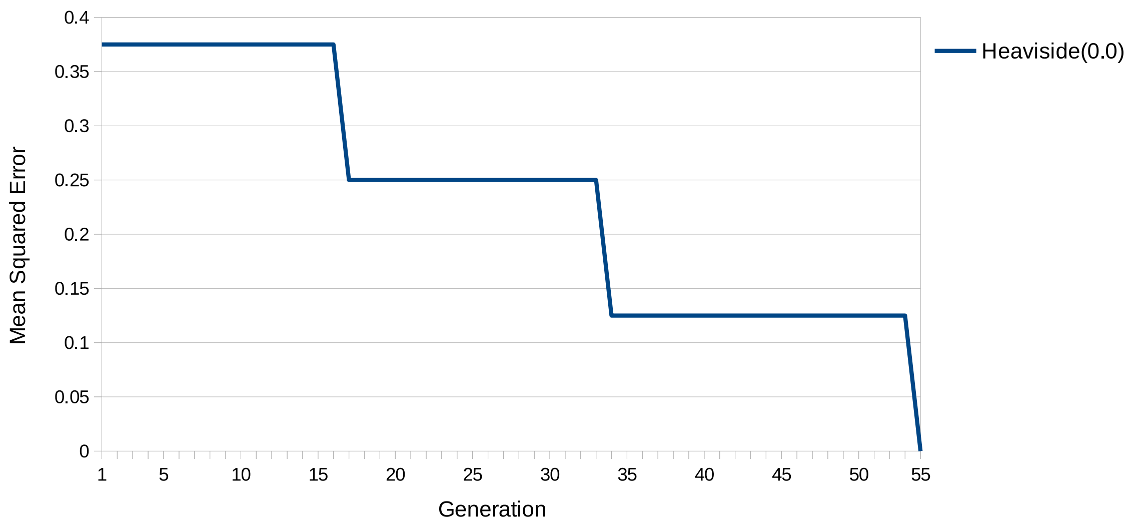


Figure 8.3: Parity of 3 inputs Cellular encoding convergence

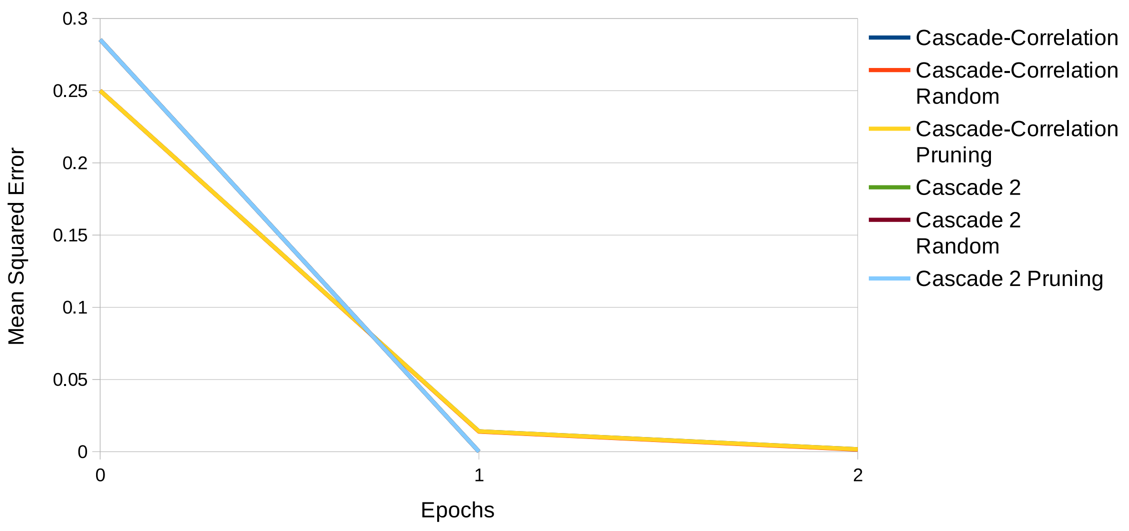


Figure 8.4: Parity of 3 inputs Cascade-Correlation and Cascade 2 convergence

Figure 8.4 shows the graph of Cascade-Correlation and Cascade 2 algorithms convergence. We can see that both algorithms are able to solve this problem Very quickly. In this example, algorithms do not profit from proposed improvements. On the other hand, backpropagation as can be seen in figure 8.5 needs hundreds to thousands steps to learn this function. These results were obtained from 30 runs by selecting the median solution compared by mean squared error. Nevertheless, the differences between solutions were negligible.

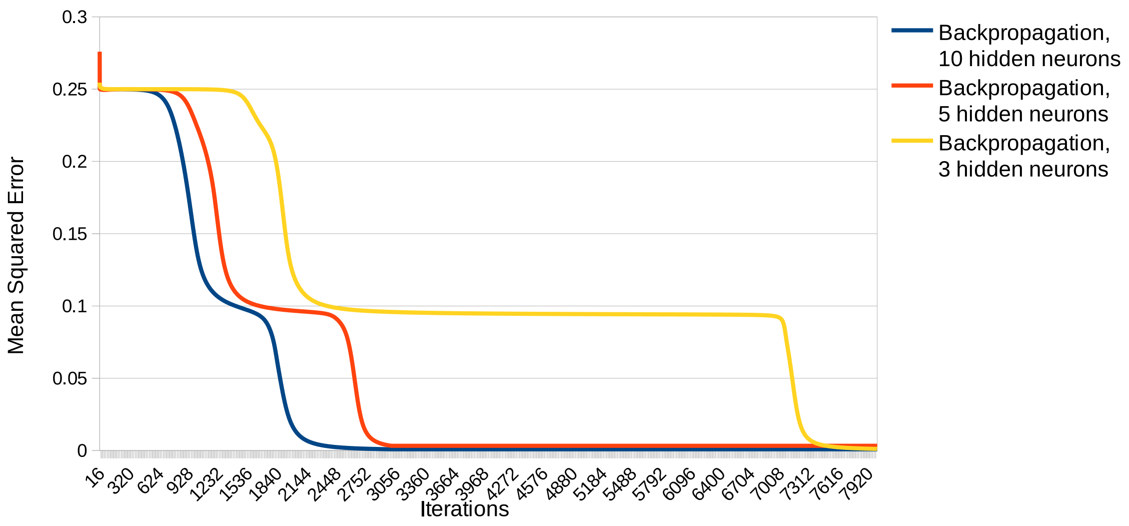


Figure 8.5: Parity of 3 inputs backpropagation convergence

8.3 Parity of 4 values

This problem is similar to the previous one, but the number of inputs is 4. This section provides results of Cascade-Correlation and Cascade 2. Results of Cellular-Encoding are not provided due to the fact, that algorithm failed to solve this problem.

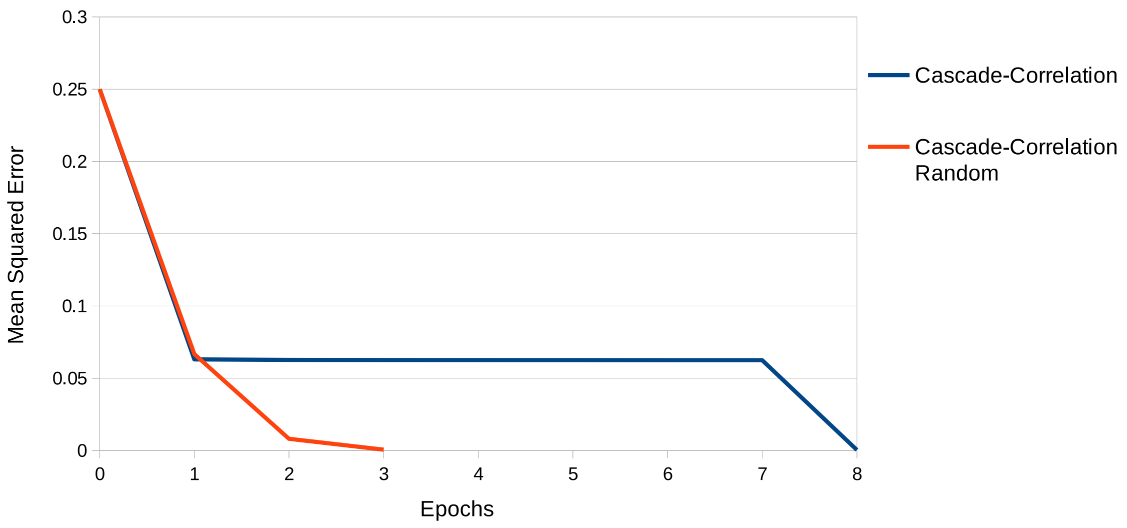


Figure 8.6: Parity of 4 inputs Cascade-Correlation and Cascade 2 convergence

In the figure 8.6, we can see the graph of Cascade-Correlation convergence. Classical version of Cascade-Correlation converges in 9 learning steps, that means 8 neurons is added to network. Proposed improvement **random search of output weights** decreases the number of learning to 4 and three neurons. Both algorithms successfully resolved the problem in all executed experiments.

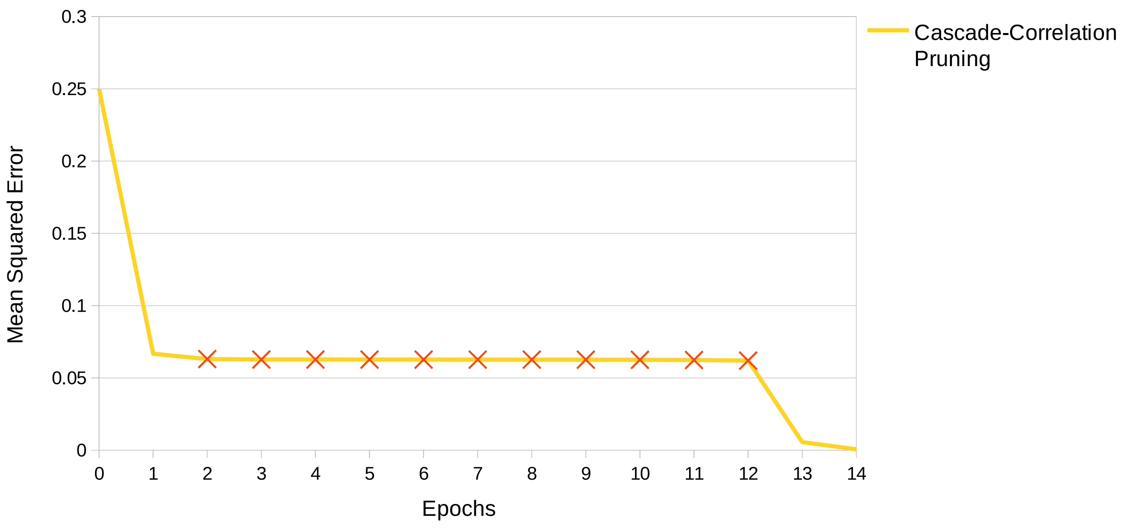


Figure 8.7: Parity of 4 inputs Cascade-Correlation pruning, red crosses represent epochs, where pruning was performed

Figure 8.7 describes the convergence of Cascade-Correlation pruning with $\epsilon = 0.98$. Red crosses represent epochs, where pruning was performed. We can see, that the number of epochs increased to 15, while the number of neurons decreased to three.

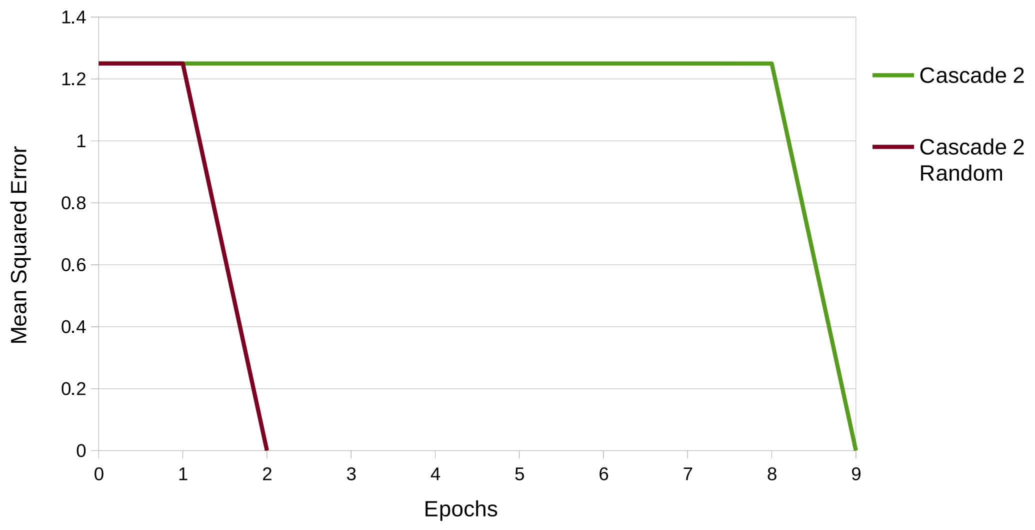


Figure 8.8: Parity of 4 inputs Cascade2 algorithm

Figure 8.8 shows convergence of Cascade 2 algorithm. This algorithm was run with input in range $\langle -1, 1\rangle$. With usage of proposed improvement **random search of output weights**, the number of epochs decreases to 3 and the number of hidden neurons to 2. When **pruning** was enabled, algorithm failed to converge in all of thirty executions.

8.4 Chess 3x3 Problem

This problem consists of classification of boxes on chessboard. For experiments, chessboard of size 3x3 had been chosen. Experiments with different number of patterns were run. First, experiments with 250 samples were executed.

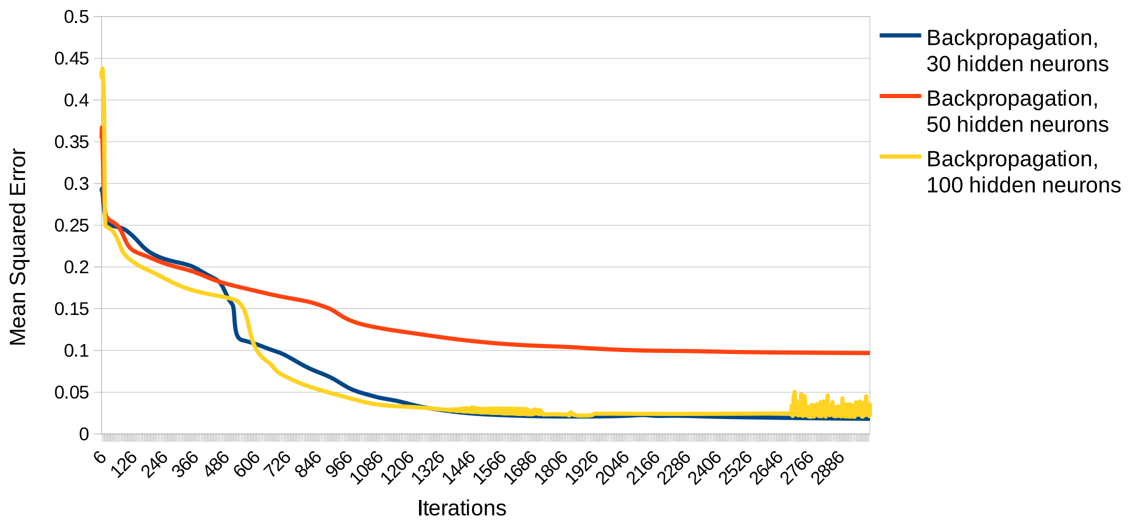


Figure 8.9: Ches 3x3 backpropagation convergence for 250 samples

In figure 8.9 we can see convergence of backpropagation. This can be compared to convergence of Cascade-correlation, that is depicted in figure 8.10. From this figure, we can see, that **random search of output weights** resulted in lower error and quicker convergence, while **pruning** made response worse.

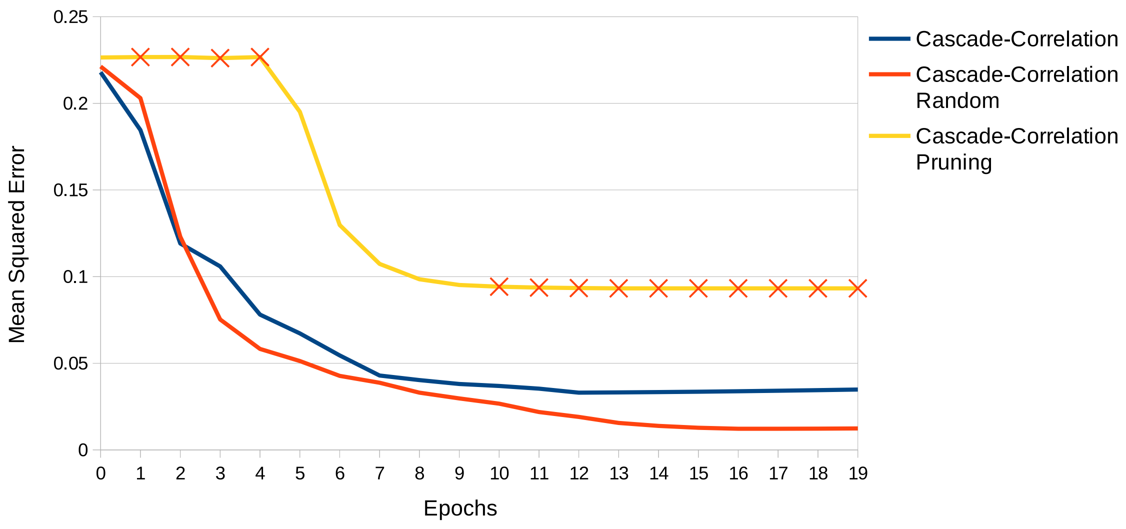


Figure 8.10: Ches 3x3 Cascade-Correlaton convergence for 250 samples

Figure 8.11 depicts response of network. We can see over-fitting on the picture of cascade-correlation, while backpropagation is more fuzzy.

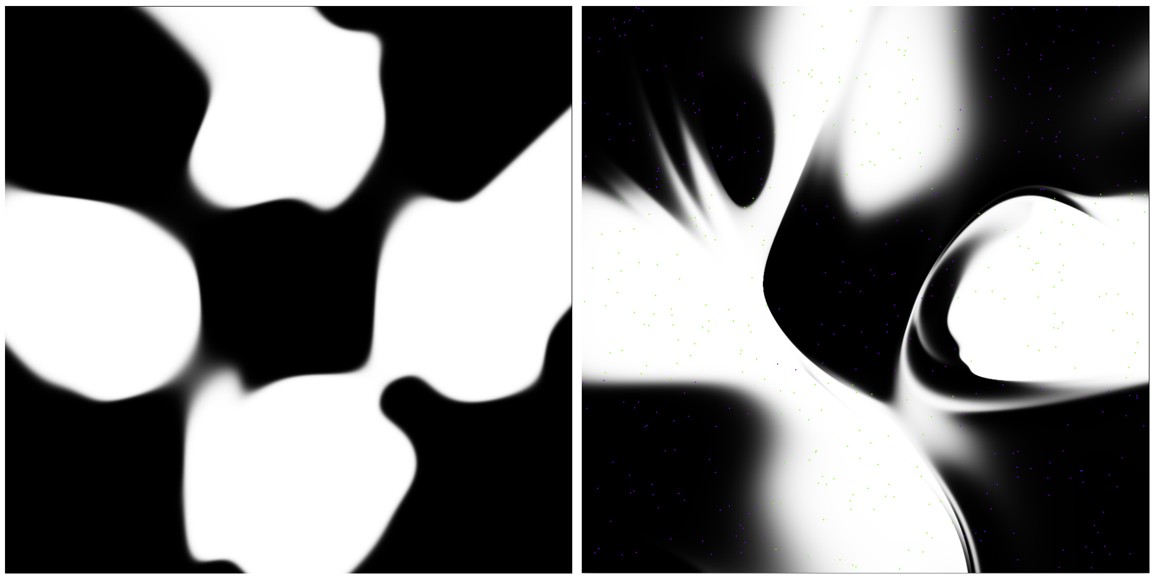


Figure 8.11: 250 samples for Chess 3x3, response of backpropagation can be seen on the left side, cellular encoding right, each axis describe one input of network, and color its class

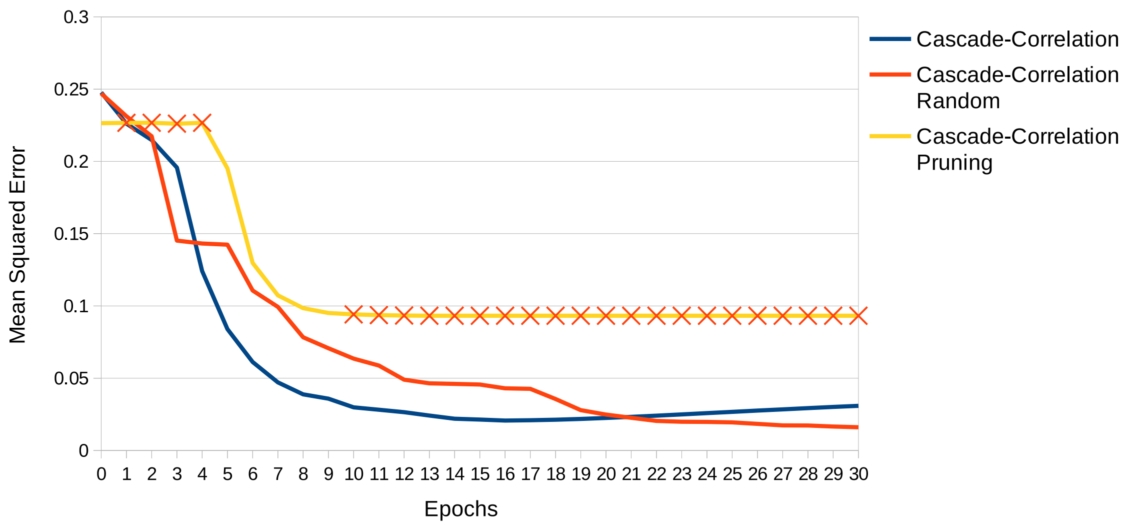


Figure 8.12: Ches 3x3 Cascade-Correlaton convergence for 1000 samples

Figure 8.12 depicts convergence of Cascade-Correlation. The speed of convergence is comparable to the speed of convergence for 250 samples. **Random search of output weights** results in lower error, but the speed of convergence is slower in the first epochs. Improvement of **pruning** fails to converge again.

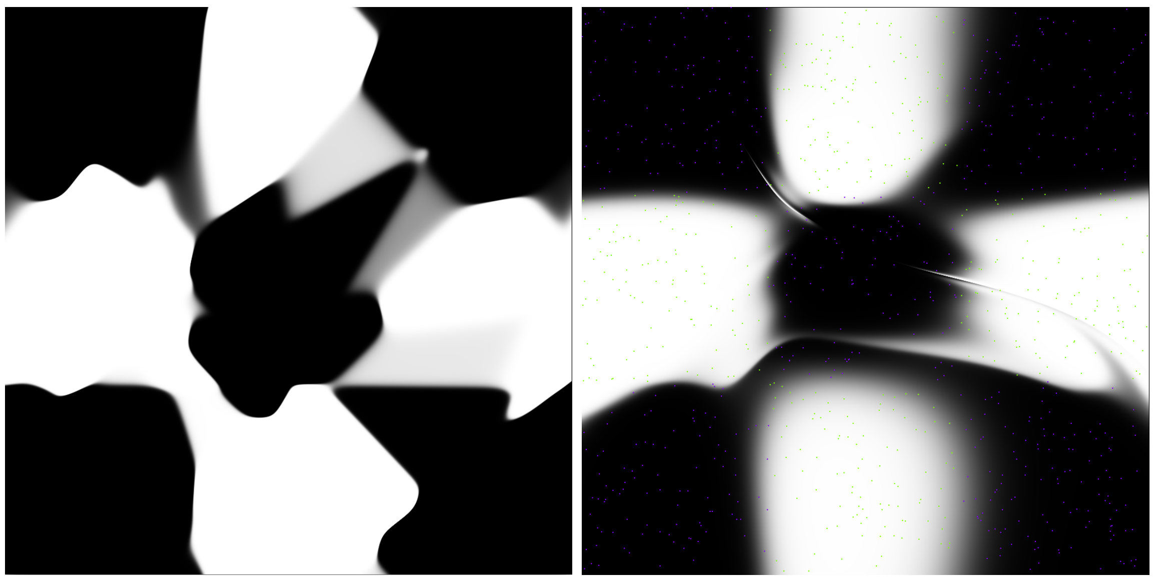


Figure 8.13: 1000 samples for Chess 3x3, response of backpropagation can be seen on the left side, cascade-correlation right, each axis describe one input of network, and color class, network classifies corresponding input to

We can see, that response of backpropagation in figure 8.13 is blurred, while response of cascade-correlation is sharper, but over-fitting is well seen.

Chapter 9

Evaluation

This chapter deals with evaluation of implemented algorithms one by one and in the end of the chapter the possibilities of further work are proposed.

Algorithm **Cascade-Correlation** gives very good results in all introduced experiments. This algorithm was able to resolve parity of 3 and 4 values and XOR very quickly with minimal topology in all executed cases. On the chess 3x3 problem, the over-fitting of network could be seen. This behavior was also observed by other authors [15].

Cascade 2 algorithm shows very promising results for XOR and parities. Even though, for chess 3x3 problem algorithm does not converge. Given the fact that algorithm was never published, it is not possible to compare this results to other works. This can indicate the fact, that the algorithm works well for bipolar inputs, but further work is needed for decision about correctness of this claim.

The first proposed improvement **random search of output weights** gives promising results in the case of Chess 3x3 and parity of 4 values, but further experiment is needed for verification of these results. This improvement decreases the number of epochs on both **Cascade-Correlation** and **Cascade 2**, but on the other hand it increases the time spent on learning.

The second proposed improvement **pruning of added neurons** decreases the number of neurons added by **Cascade-Correlation**, but makes the algorithm less stable. In the case of **Cascade 2**, this improvement shows unpredictable behavior and I would not recommend to use it.

Cellular-Encoding is a promising way for developing artificial neural networks, but needs extra work. I was not able to obtain the same results as described in the original thesis due to the fact, that genetic programming configuration was not precisely described in the original dissertation. I see the necessity of setting wide number of configurations for this algorithm as a big deficit. The algorithm is dependent on configuration of genetic programming and selected instruction set. The convergence when using a wider – complete instruction set is much slower and the rate of success in problem solving dramatically decreases. Algorithm suffers next to these two deficiencies by the way it develops the number of input and output units. This number is given by cellular code and can't be hardcoded. The algorithm works relatively well, when these parameters are well tuned, but given the complexity of configuration, I would not recommend this algorithm as production-ready.

Chapter 10

Conclusion

The goal of this work was the implementation of selected algorithms, exploration of possible improvements and evaluation of algorithms. This goal was achieved by implementing algorithms in C++ language.

The theoretical part of this thesis describes the fundamentals of neural networks, mathematical models of neurons and networks. Further it presents basic algorithms for learning neural networks and explores algorithms for topology of neural networks development. Algorithms are divided into three groups – pruning, constructive and neuroevolutionary. Pruning algorithms start with a large network and remove neurons and connections to make them smaller. On the other hand constructive algorithms start with small networks - usually without hidden neurons and add neurons and weights. Neuroevolutionary algorithms develop neural networks by using evolutionary algorithm, for example genetic programming or genetic algorithm.

The practical part deals with description of concrete algorithm implementation. One of the selected algorithms, **Cascade 2** was never published. These algorithms were implemented as a part of the existing library **NeuralNetworkLib**. Besides these algorithms, a new library for genetic programming was introduced.

The last part of the thesis describes performed experiments with these algorithms. It compares these algorithms with each other and shows influence of proposed improvements. These experiments show that one algorithm responds well to all performed experiments – **Cascade-Correlation**. The algorithm **Cascade 2** provides nice results on bipolar experiments. The last selected algorithm – **Cellular encoding** works on only one experiment. Possible reasons for this are described in the evaluation chapter.

Two proposed improvements **random search of output weights** and **pruning of added neurons** decrease the number of used neurons, but on the other hand, the time for training is increased.

Further possibility of extensions could be expansion of instruction set for Cellular encoding or solving its problem of hard-coded sizes of inputs and outputs. Another possible extension lies in implementing more constructive algorithms and comparing them.

Bibliography

[1] Caianiello, E. R.; Orlandi, G.; Piazza, F.; et al.: Dynamic Pruning In Artificial Neural Networks. 1991.

[2] Chiou, Y.; Lurecorrespondence, Y.: Hybrid Lung Nodule Detection (HLND) system. In *Cancer Letters*. 1994. pp. 119–126. doi:doi:10.1016/0304-3835(94)90094-9.

[3] Fahlman, S. E.; Boyan, J. E.; Baker, D.: The cascade 2 learning architecture (UNPUBLISHED). In *Technical Report CMU-CS-TR-96-184*. Carnegie Mellon University. 1996.

[4] Fahlman, S. E.; Lebiere, C.: The cascade-correlation learning architecture. In *Advances in Neural Information Processing Systems 2*. Morgan Kaufmann. 1990. pp. 524–532.

[5] Frean, M. R.: A „Thermal" Perceptron Learning Rule. *Neural Computation*. vol. 4, no. 6. 1992: pp. 946–957.

[6] Golea, M.; Marchand, M.: A Growth Algorithm for Neural Network Decision Trees. *EUROPHYSICS LETTERS*. vol. 12. 1990: pp. 205–210.

[7] Gruau, F.; lyon I, L. C. B.; Doctorat, O. A. D. D.; et al.: Neural Network Synthesis Using Cellular Encoding And The Genetic Algorithm. 1994.

[8] Hecht-Nielsen, R.: Neural Networks for Perception (Vol. 2). chapter Theory of the Backpropagation Neural Network. Orlando, FL, USA: Harcourt Brace & Co.. 1992. ISBN 0-12-741252-2. pp. 65–93.

[9] Islam, M. M.; Sattar, M. A.; Amin, M. F.; et al.: A New Constructive Algorithm for Architectural and Functional Adaptation of Artificial Neural Networks. 2009.

[10] Koza, J. R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press. 1992. ISBN 0-262-11170-5.

[11] Kárný, M.; Warwick, K.; Kùrková, V.: The Psychological Limits of Neural Computation. In *Dealing with Complexity*. Perspectives in Neural Computing. Springer London. 1998. ISBN 978-3-540-76160-0. pp. 252–263. doi:10.1007/978-1-4471-1523-6_17.

[12] McCulloch, W.; Pitts, W.: A Logical Calculus of Ideas Immanent in Nervous Activity. In *Bulletin of Mathematical Biophysics*. 1943. ISBN 0007-4985. pp. 115–133. doi:doi:10.1007/BF02478259.

[13] Michel, O.; Herrmann, A.: Perceptron Learning Algorithm. 1999.
Retrieved from:
`http://lcn.epfl.ch/tutorial/english/perceptron/html/learning.html`

[14] Nissen, S.: Large Scale Reinforcement Learning using Q-SARSA(λ) and Cascading Neural Networks. 2007.

[15] Ondráček, T.; učení technické v Brně. Fakulta informačních technologií, V.: *Adaptivní vícevrstvé neuronové sítě*. Vědecké spisy: PhD Thesis. Vysoké učení technické. 2006. ISBN 9788021431263.

[16] Piszcz, A.; Soule, T.: A Survey of Mutation Techniques in Genetic Programming. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*. GECCO '06. New York, NY, USA: ACM. 2006. ISBN 1-59593-186-4. pp. 951–952. doi:10.1145/1143997.1144165.

[17] Poli, R.; Langdon, W. B.: Genetic Programming with One-Point Crossover and Point Mutation. In *Soft Computing in Engineering Design and Manufacturing*. Springer-Verlag. 1997. pp. 180–189.

[18] Rosenblatt, F.: *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory. 1957.

[19] Stanley, K. O.; Miikkulainen, R.: Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*. vol. 10: page 2002.

[20] Subirats, J. L.; Franco, L.; Jerez, J. M.: C-Mantec: A novel constructive neural network algorithm incorporating competition between neurons. *Neural Networks*. vol. 26. 2012: pp. 130–140. doi:10.1016/j.neunet.2011.10.003.

[21] Svozil, D.; Kvasnička, V.; Pospíchal, J.: Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*. vol. 39, no. 1. 1997: pp. 43 – 62. ISSN 0169-7439.

[22] Walker, M.: Introduction to Genetic Programming. 2001.
Retrieved from:
`https://www.cs.montana.edu/~bwall/cs580/introduction_to_gp.pdf`

Appendices

List of Appendices

Appendix A

Contents of attached CD

- **thesis_print.pdf** – pdf of thesis for printing
- **thesis_hyper.pdf** – pdf of thesis for viewing
- **src/** – source codes for programs
  - **lib/** – folder with library **NeuralNetworkLib**
  - **README** – file describing how to compile and run programs
  - **cascade2.cpp** – Cascade 2 experiments
  - **cascadecor.cpp** – Cascade Correlation experiments
  - **cellular_encoding.cpp** – Cellular encoding experiments
  - **setup.sh** – setup for configuration and building
- **doc/** – source code for this paper

Appendix B

Manual

B.1 Installation

Program needs to be compiled, since it is written in language C++. The prerequisites for compilation are:

- `cmake` version 3.2+
- `g++` with c++14 support, tested on 5.2.1
- `OpenGL`
- `GLUT Library`

When software is installed, run script `setup.sh`. This script compiles all necessary source codes and builds programs in folder build. If argument `-f` is provided, it downloads all libraries that are attached again.

B.2 Usage

This work consists of 3 programs – `cascade2`, `cascadecor`, `cellular_encoding`. These programs allow to run experiments. When argument `-h` is passed, they provide documentation. If programs are run without argument, they execute experiment with XOR.

Appendix C

Cellular code for XOR

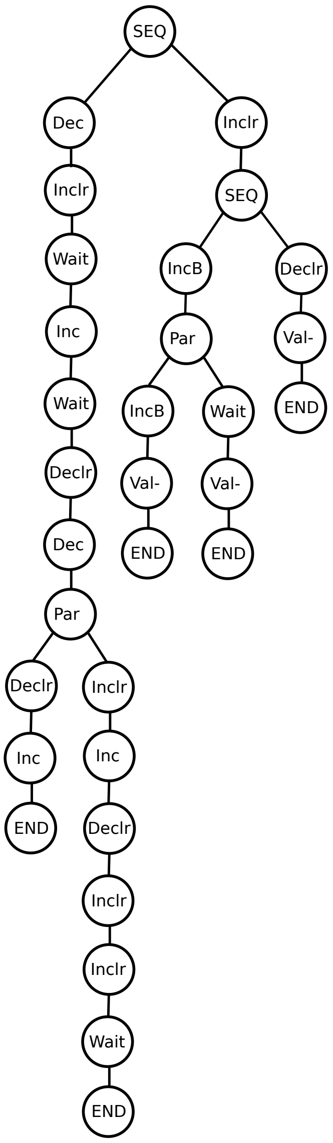


Figure C.1: XOR tree