# Modeling the Bank Client's Behavior with LTF-C Neural Network

Marcin Wojnarski

Institute of Informatics, Warsaw University,
Banacha 2, 02-097 Warsaw, Poland
mwojnars@ns.onet.pl

**Abstract.** This paper describes an application of Local Transfer Function Classifier (LTF-C) to recognition of active and non-active bank clients, which was the problem of the 2nd EUNITE Competition.
LTF-C is a neural network solving classification problems. It has similar architecture as the Radial Basis Function neural network, but utilizes entirely different training algorithm. This algorithm is composed of: changing positions and sizes of reception fields of hidden neurons, insertion of new hidden neurons and removal of unnecessary ones during the training. LTF-C was chosen to solve this problem, because it had performed very well in other real-world problems, such as handwritten digit recognition, credit risk assessment or classification of breast cancer tissue.
The modeling of the bank client's behavior was performed in three stages. First, the data were preprocessed: nominal values were changed to numerical, every attribute was rescaled and transformed in order to equalize its histogram. Then, several tens of neural networks were trained. Finally, a committee of the best 14 networks was created.
The paper presents also some possible directions of further research, which could lead to the increase of the usefulness and effectiveness of the system.

## 1 Introduction

This paper describes an application of Local Transfer Function Classifier (LTF-C) to recognition of active and non-active bank clients, which was the problem of the 2nd EUNITE Competition. Theoretical foundations of LTF-C are presented with details in the next chapter.

The data prepared for the competition were difficult to handle for LTF-C, so they required thorough preprocessing: nominal values were changed to numerical, every attribute was rescaled and transformed in order to equalize its histogram.

The datasets were relatively large – each set consisted of 12000 patterns. Despite large amount of data available, the classification problem was difficult to solve – it was impossible to obtain the error rate on the training set significantly lower than 20% (while preserving good generalization, of course). In order to classify the test patterns as well as possible, a committee of 14 networks was created.

## 2 Local Transfer Function Classifier

Local Transfer Function Classifier (LTF-C) [1] is a neural network designed for solving classification problems. After training it is able to recognise – when presented a pattern $X$ ($X = [x_1, x_2, \ldots, x_n]$) – which class $c$ ($c = 1, 2, \ldots, k$) this pattern belongs to.

LTF-C has almost the same architecture as the Radial Basis Function (RBF) neural network [2, 3], but it employs new training algorithms. The most noticeable difference is that the structure of LTF-C is not defined at the beginning of the training, as in the case of most of other neural systems, but changes dynamically during the learning. Such an algorithm allows to fit better to the training set and guarantees that the network will be just as big, as it is really needed.

### 2.1 The Network Architecture

Let the training set be composed of $N$ pairs of the form: $(X^{(i)}, c^{(i)})$. Vectors $X^{(i)}$ can be treated as points in $n$-dimensional space $\mathbf{X}$ (we can identify vectors with points, so for the simplicity of the notation these terms will be used interchangeably). Close neighborhood of the point $X^{(i)}$ should belong to the same class as $X^{(i)}$, therefore the space $\mathbf{X}$ can be divided into finite number of *decision regions* – areas of the same value of classification. The classification problem resolves then to the task of modeling complex figures in $n$-dimensional space. One of possible solutions to such a task is to model interiors of these regions – by filling them as tight as possible with figures of versatile shapes. This idea lays in the basis of LTF-C.

The network is composed of two layers of neurons. The first one retains the information about figures filling the decision regions. Each figure is represented by a neuron (forms its *reception field*): the neuron *weights* define the position of the figure centre, and the *radii* – its size. The neuron output, belonging to the range of $[0, 1]$, says how "much" the presented pattern lies in the interior of the figure. Formally, the response $y_i$ of the $i$-th neuron on the pattern $X$ is given as:

$$y_i = f\left(\sqrt{\sum_{j=1}^{n}\left(\frac{w_{ij} - x_j}{r_{ij}}\right)^2}\right), \tag{1}$$

where $W_i = [w_{i1}, w_{i2}, \ldots, w_{in}]$ – weights of the $i$-th neuron, $R_i = [r_{i1}, r_{i2}, \ldots, r_{in}]$ – radii of the $i$-th neuron, $f$ – an output function. With such a definition reception fields have shapes of hyperellipses with axes parallel to the axes of the co-ordinate system.

Reception fields have to fill some – the most often bounded – region, so they have to be bounded figures themselves. Hence the output function should satisfy:

$$\lim_{d\to\infty} f(d) = 0\ , \tag{2}$$

which guarantees that the transfer function will be *local* – neuron responses will vanish for points $X$ lying far from $W_i$. Usually the Gaussian function is used as

$f$, yielding the following value of the neuron response:

$$y_i = \exp\left(-\sum_{j=1}^{n}\left(\frac{w_{ij}-x_j}{r_{ij}}\right)^2\right). \tag{3}$$

Every hidden neuron must remember what is the class of the decision region it fills. To this end, it uses weights of connections with output neurons (they are not being modified during the training). If the $i$-th hidden neuron belongs to the $c$-th class (fills the decision region of the $c$-th class) the weight $w'_{ij}$ of its connection with the $j$-th output neuron equals:

$$w'_{ij} = \begin{cases} 0 & \text{for } j \neq c \\ 1 & \text{for } j = c \end{cases}. \tag{4}$$

The output layer just aggregates the information coming from the hidden one. It is composed of $k$ neurons ($k$ – the number of classes) – if the $i$-th neuron is the most activated one after the presentation of the pattern $X$, it means that the network has classified $X$ to the $i$-th class. This layer is composed of simple linear units – a response $y'_i$ of the $i$-th output neuron equals:

$$y'_i = \sum_{j=1}^{m} w'_{ji} y_j \ , \tag{5}$$

where $m$ is the number of hidden neurons.

### 2.2 Changing Position of Reception Fields

The goal of a hidden neuron belonging to the $c$-th class is to position its reception field in such a way that it contains as much points from the $c$-th class and as little points from other classes as possible. For that reason, during the training phase the neuron should move its weights $W$ towards the points $X$ belonging to the $c$-th class and move away from the ones belonging to other classes. Moreover, the higher the neuron response on the presented pattern, the greater the influence of that pattern on modification of the neuron weights should be. Thus, a new value of weights of the $i$-th hidden neuron, belonging to the $c_i$-th class, after presentation of the pattern $X$ from the $c$-th class, should be a weighted mean of their previous value and $X$:

$$W_i \leftarrow W_i + \eta y_i (X - W_i) \ , \tag{6}$$

$$\eta = \begin{cases} \eta^+ & \text{for } c_i = c \\ -\eta^- & \text{for } c_i \neq c \end{cases}, \tag{7}$$

where $\eta^+$ and $\eta^-$ are constants ($0 < \eta^+ \leq \eta^- \leq 1$). Usually $\eta^+ = \eta^- = 1$.

However, learning according to (6) has some disadvantages. Hidden neurons are trained entirely independently, therefore they will gather after training in

several regions of the input space – these ones, where the concentration of training points is the largest. In other regions of the input space there will not be any neurons. Another disadvantage of this formula is that differences in the difficulty of classification in various parts of the input space are not taken into account, while more neurons are needed in regions of more complicated decision borders. To solve these problems the term of the *attractiveness* of the learning pattern was introduced. It defines how big influence on the modification of weights specified learning pattern should have. The worse (less correct) the network response on the pattern $X$, the bigger the attractiveness of this pattern should be.

Before giving the definition of the attractiveness we must say what we mean by less or more correct response. *Correctness* $\Delta$ of the network response on the pattern $X$ from the $c$-th class was defined as follows:

$$\Delta = y'_c - \max\{\, y'_i : i \neq c \,\} \ . \tag{8}$$

The sign of $\Delta$ says whether the network answer was correct, and its absolute value says how sure the network was while giving this response. Certainly, the greater $\Delta$ the more correct the answer of the network is.

The attractiveness function $A(\Delta)$ must satisfy the following conditions:

1. $A(\Delta) \in [0, 1]$ – For the learning process to be stable.
2. $\lim\limits_{\Delta \to +\infty} A(\Delta) = 0$ – Well classified patterns do not influence learning then.
3. $\lim\limits_{\Delta \to -\infty} A(\Delta) = 0$ – It ensures that patterns mistakenly classified during the data acquisition will not have significant influence on the training. We can observe a similar property in the way people acquire knowledge – if the human get information completely unfitting his current knowledge, he does not believe in it, e.g. presuming he has misheard. Only if the same information come at him several times, he adjusts his opinion. This feature is also needed in the case of overlapping decision regions.

Taking into account above conditions, $A(\Delta)$ was defined as a Gaussian function with two slopes of separately chosen biases:

$$A(\Delta) = \begin{cases} \exp\left(-2\left(\frac{\Delta - \Delta_0}{\Delta_{\max} - \Delta_0}\right)^2\right) & \text{for } \Delta \geq \Delta_0 \\ \exp\left(-2\left(\frac{\Delta - \Delta_0}{\Delta_{\min} - \Delta_0}\right)^2\right) & \text{for } \Delta < \Delta_0 \end{cases} , \tag{9}$$

where $\Delta_0$, $\Delta_{\min}$, $\Delta_{\max}$ are constants satisfying: $\Delta_{\min} < \Delta_0 \leq 0$ and $\Delta_0 < \Delta_{\max}$. Usually: $\Delta_0 = -0.5$, $\Delta_{\min} = -1.0$, $\Delta_{\max} = 0.5$.

There is still one more disadvantage of (6). The range of modifications of the reception field position does not depend on the number of learning steps to carry out. Thus, even just before the end of the training the modifications are large, preventing neurons from fitting well to the data. To correct this drawback a parameter $\tau_i^{(t)}$ was introduced, which denotes the velocity of the training of the $i$-th hidden neuron in the $t$-th training step:

$$\tau_i^{(t)} = 1 - \frac{t - t_i}{T - t_i} \ , \tag{10}$$

where $t_i$ – a training step when the $i$-th neuron was created, $T$ – the total number of training steps to carry out. The parameter $\tau_i$ is decreasing linearly from 1 at the moment of creating the $i$-th neuron to 0 in the last training step.

And the corrected formula for the weights modification has the form:

$$W_i \leftarrow W_i + \eta \tau_i^{(t)} A(\Delta) y_i (X - W_i) \ . \tag{11}$$

### 2.3 Changing Size of Reception Fields

Size of the neuron reception field is defined by the vector of radii $R_i$ (1), independently along each axis of the co-ordinate system. One of reasons for adjusting it adaptively is that regions of different size and difficulty of classification can exist in the input space simultaneously. There can exist, for instance, vast areas of univocal classification, very easy to model with only one huge reception field, and regions adjacent to decision borders, requiring high precision and, therefore, small reception fields. Another reason is that different attributes can be of unequal importance for classification – some of them can be insignificant, corresponding radii should be then large, while others can play the vital role in classification – corresponding radii ought to be quite short.

Change of the $j$-th radius of the $i$-th neuron after presentation of the sample $(X, c)$ should depend on:

1. the response $y_i$ of the neuron – in order to allow only patterns in the reception field to influence the training,
2. the attractiveness of the pattern – to enable difficult patterns to have bigger influence on the training (see ch. 2.2),
3. the number of training steps to carry out – for neurons to fit well to the data at the end of the training (see ch. 2.2),
4. the distance $d_{ij}$ along the $j$-th axis between the pattern and the centre of the reception field:
$$d_{ij} = \left| \frac{x_j - w_{ij}}{r_{ij}} \right| . \tag{12}$$

The following formula satisfying given assumptions was devised:

$$r_{ij} \leftarrow r_{ij} \exp\left( \eta_{\mathrm{g}} \tau_i^{(t)} A(\Delta) y_i d_{ij} \right) \ , \tag{13}$$

$$\eta_{\mathrm{g}} = \begin{cases} \eta_{\mathrm{g}}^{+} & \text{for } c_i = c \\ -\eta_{\mathrm{g}}^{-} & \text{for } c_i \neq c \end{cases} \ , \tag{14}$$

where $c_i$ is the number of a class which the $i$-th neuron belongs to, and $\eta_{\mathrm{g}}^{+}$ and $\eta_{\mathrm{g}}^{-}$ are constants ($0 < \eta_{\mathrm{g}}^{+} \leq \eta_{\mathrm{g}}^{-}$, usually $\eta_{\mathrm{g}}^{-} = 1$ and $\eta_{\mathrm{g}}^{+} = 0.8$).

### 2.4 Inserting Hidden Neurons

Before starting training neurons, they have to be created, with weights and radii properly initialized. It is no that easy – when adaptive parameters are initialized

randomly, nearly all reception fields land in regions with no training points. Initializing centres of reception fields with random points from the training set is not good, as well, as most of the neurons will be in regions, where many points lie, not where difficult classification requires more units. The best solution – applied in LTF-C – is adding neurons during the training, in regions where the network response is unsatisfactory.

In the $t$-th step of the training, after presentation of the sample $(X, c)$, a neuron is inserted to the hidden layer with probability $P$, depending on $A(\Delta)$ (9), i.e. on how incorrect the network response has been, and $\tau_{\text{ins}}^{(t)}$, saying how intensive the process of creating new neurons should be in that learning step (compare with $\tau_i^{(t)}$ in ch. 2.2):

$$P = \tau_{\text{ins}}^{(t)}\, pA(\Delta) \ , \tag{15}$$

where $p$ is a positive constant (most often $p = 0.05$) and $\tau_{\text{ins}}^{(t)}$ is defined as:

$$\tau_{\text{ins}}^{(t)} = \begin{cases} 1 - \dfrac{t}{0.9T} & \text{for } t < 0.9T \\ 0 & \text{for } t \geq 0.9T \end{cases} \ . \tag{16}$$

In the last 10% of time $\tau_{\text{ins}}^{(t)} = 0$, since neurons created just before the end would not have enough time to learn.

Weights of the inserted neuron are initialized as follows ($m$ – the number of hidden neurons existing till now):

$$W_{m+1} = X \ , \tag{17}$$

$$w'_{(m+1)i} = \begin{cases} 0, & \text{for } i \neq c \\ 1, & \text{for } i = c \end{cases} \ , \tag{18}$$

where $w'_{(m+1)i}$ is the weight of the connection with the $i$-th output neuron.

Initializing radii is more difficult. They should be rather long, as even one too small radius may result in excluding all the training points from the reception field. However, they should not be also too large either, since a new neuron could disturb the training process of other units too much. The following formula satisfies above conditions quite well:

$$r_{(m+1)i} = r_{\min} + Y(r_{\max} - r_{\min}) \ , \tag{19}$$

where:

$$r_{\min} = \min S \ , \quad r_{\max} = \max S \ , \tag{20}$$

$$S = \{\, r_{ij} : 1 \leq i \leq m, 1 \leq j \leq n \,\} \cup \left\{ \frac{\sqrt{n}}{10} \right\} \tag{21}$$

(value of $\frac{\sqrt{n}}{10}$ was picked out under the assumption that components $x_i$ of input vectors belong to $[0, 1]$, more or less), $Y$ – a random variable of uniform distribution from the range of $[0, 1]$. Each component of the vector $R_{m+1}$ is initialized individually.

### 2.5 Removing Hidden Neurons

Despite a sophisticated algorithm for creating neurons, many of them land in regions where they are useless or even harmful, only worsening the network performance. Thus, an algorithm for removing unnecessary hidden neurons is needed. The one used in LTF-C evaluates after each presentation of a pattern so-called *global usefulness* $u_i$ of every hidden neuron. For this purpose it utilizes *instantaneous usefulness* $v_i$, saying how important the existence of the $i$-th neuron has been for reckoning a correct network response on only one pattern $X$. The instantaneous usefulness is computed only on the ground of the last presented sample $(X, c)$, according to the formula:

$$v_i = A(\Delta_i) - A(\Delta) \ , \tag{22}$$

where $A$ is the attractiveness function (9), $\Delta$ – correctness of the last response of the network, and $\Delta_i$ says how correct the response would have been if the $i$-th neuron had not existed (compare (8) and (5)):

$$\Delta_i = y_c^{(i)} - \max\left\{ y_k^{(i)} : k \neq c \right\} \ , \tag{23}$$

$$y_j^{(i)} = y'_j - w'_{ij} y_i \ . \tag{24}$$

The instantaneous usefulness $v_i$ is positive if the $i$-th neuron has had beneficial contribution to reckoning the network response, and negative if the response would have been better after removing this neuron. Evaluating $v_i$ for all neurons is not very expensive – complexity of this operation is proportional to the number of weights of the output layer.

The global usefulness $u_i$ of the $i$-th neuron should be an average of values $v_i$ computed for different training patterns. Arithmetic mean of $v_i$ for all samples would be the best, but its use is impossible due to high memory requirements and computational complexity. Therefore, exponential mean was applied – only last values of $u_i$ and $v_i$ are necessary to calculate it. One has only to remember that patterns must be presented in each epoch in a different order, since this sequence influences the value of the usefulness. The formula for modification of $u_i$ has the form:

$$u_i \leftarrow (1 - \eta_\mathrm{u}) u_i + \eta_\mathrm{u} v_i \ , \tag{25}$$

where $\eta_\mathrm{u}$ is a constant from the range of $[0, 1]$. The $i$-th neuron is removed when

$$u_i < U \ , \tag{26}$$

where the threshold $U$ is a constant: $U \in [0, 1]$. Usually $U \approx \eta_\mathrm{u}$.

The requirement that an exponential mean with the parameter $\eta_\mathrm{u}$ should have similar properties as an arithmetic mean of $N$ components ($N$ – size of the training set) yields that $\eta_\mathrm{u}$ should be approximately $\frac{2}{N}$. And imagining what should happen with a neuron which reception field does not contain any training points yields that during neuron creation the usefulness $u_i$ should be initialized with the value of $e^2 U \approx 8U$.

## 3 Data Preprocessing

The data prepared for the competition were composed of 24000 patterns, 12000 in the training set and 12000 in the test set. Every pattern consisted of 36 attributes, 6 nominal and 30 numerical. Each training pattern was also followed by the information about its class (there were two classes: "_0_" and "_1_"). The training set contained the same number of patterns from both classes.

The data prepared for the competition were difficult to handle for LTF-C, so they required thorough preprocessing.

### 3.1 Turning Nominal Attributes to Numerical

First six attributes were originally nominal. They had to be turned to numerical, since LTF-C cannot handle nominal attributes.

This transformation was easy to perform, as these attributes were in fact numerical. They were composed of a natural number preceded and followed by an underscore ("_"). The only thing to do was removing the underscores. Attributes obtained in this way had similar histograms as other attributes (see the next chapter), which proved that this operation was justified.

### 3.2 Rescaling and Histogram Equalization

Magnitude of input values differed very much from one attribute to another. E.g., there was an attribute of extremely small magnitude – of the order of 1.0e-36. Since the training algorithm of LTF-C requires some kind of normalization of every attribute (the dispersion should not be neither too big nor too small), the inputs had to be rescaled.

Typical method of rescaling is to divide the attribute either by its standard deviation or by the difference between maximum and minimum value. However, this method failed for the competition data, because deviation was usually very small, and $max - min$ was very big. This was due to atypical distribution of the values in every attribute, far from normal distribution.

In order to find a good method and coefficients of normalization I had to make thorough analysis of histograms of every attribute. This analysis revealed that almost every attribute had a value – let us denote it by $x_0$ – appearing very often, typically in about 75% of patterns. And over 90% of values lay in the very close neighborhood of $x_0$ (let us denote the width of this neighborhood by $\epsilon$). So the distribution was always concentrated in a single point – that is why the standard deviation was so small.

On the other hand, there were also values very far from $x_0$, e.g., $x_0 + 1000\epsilon$. That is why $max - min$ was so big. Although very small part of values lay far from $x_0$ (no more than several per cent), they were important for performing correct classification. As the statistical analysis revealed, the existence of such a value in a pattern significantly increased the likelihood that the pattern belonged to class "_0_".

In order to make such atypical distributions more similar to standardized normal distribution, rescaling and histogram equalization was applied to every attribute. Histogram equalization was performed by finding the logarithm of an attribute. Thus, the normalization had the form:

$$x' = (x - x_0)/\epsilon \ , \tag{27}$$

$$x'' = 0.5\,\mathrm{sgn}(x')\log_{10}(|x'| + 1) \ . \tag{28}$$

Parameters $x_0$ and $\epsilon$ were chosen individually for each attribute.

### 3.3 Augmentation of the Attributes Set

Statistical analysis revealed that large values in an input pattern (*after* normalization) increased the likelihood that the pattern belonged to class "_0_". Thus, the sum or mean of attributes should have had good discriminative value and appending it into the input vector should have improved the neural network training.

This is the reason why three new attributes were added to input vectors: arithmetic mean of attributes, arithmetic mean of absolute values of attributes and quadratic mean of attributes:

$$x_{37} = \frac{1}{36}\sum_{i=1}^{36} x_i \ , \tag{29}$$

$$x_{38} = \frac{1}{36}\sum_{i=1}^{36} |x_i| \ , \tag{30}$$

$$x_{39} = \sqrt{\frac{1}{36}\sum_{i=1}^{36} x_i^2} \ . \tag{31}$$

Most networks were trained with the augmented set of attributes, but also some networks were trained with original 36 attributes.

## 4 Neural Networks Training

The training of an LTF-C neural network requires setting values of about ten training parameters (see ch. 2). Although there are some good general rules for doing this, they do not always give best results in the terms of the error rate. For that reason, I trained several tens of neural networks using different values of training parameters and created a committee [4] of the best 14 networks. The values of parameters used are given below ($T$ is the number of training steps;

other parameters are described in ch. 2):

$$\begin{aligned}
&\eta^+ = \eta^- = \eta_{\mathrm{g}}^- = 1 \ , \\
&\eta_{\mathrm{g}}^+ \in [0.8, 0.95] \ , \\
&\Delta_0 \in [-1.0, -0.25] \ , \\
&\Delta_{\min} \in [-3.0, -0.6] \ , \\
&\Delta_{\max} \in [0.3, 2.0] \ , \\
&p \in [0.01, 0.02] \ , \\
&\eta_{\mathrm{u}} \in [0.3\text{e-}4, 1.0\text{e-}4] \ , \\
&U = 1.0\text{e-}4 \ , \\
&T = 240000 \ \text{ or } \ T = 360000 \ .
\end{aligned} \tag{32}$$

The training of a single network took about 30 seconds on AMD Duron 700 MHz.

Evaluation of trained networks was difficult, because there was no single and reliable indicator of the network performance. Low error rate on the training set alone did not guarantee low error rate on the test set, due to possible poor generalization. That is why three factors were taken into account while choosing the best networks to the committee:

1. low error rate on the training set,
2. small number of hidden neurons, which guaranteed good generalization,
3. equal division of the test set into both classes. The only information about the test set was that it contained the same number of patterns from both classes (this fact could have been deduced from the information published on the competition web page). So a good network, when tested on the test set, should have given approximately the same number of classifications into classes "_0_" and "_1_".

The best 14 networks were chosen to the committee. Their error rate on the training set varied from 18.8 to 22.0% (20.3% on average). They comprised from 53 to 88 hidden neurons (898 in total, 64 on average). The percentage of patterns from the test set classified as "_1_" varied from 44.6 to 48.6% (46.4% on average).

Every network in the committee could have "voted" either for "_0_" or "_1_". The votes of all networks were counted. If at least 9 networks voted for "_0_", this was the response of the committee. Otherwise, the response was "_1_".

The choice of 9 as a threshold requires an explanation. Normally, a half of votes (i.e., 7 or 8) for "_0_" should be enough to respond "_0_". However, in that case the committee would have classified as "_1_" only 45.2 or 47.7% of test patterns, while exactly a half of test patterns belonged to class "_1_". When 9 was the threshold, exactly 6002 test patterns out of 12000 were classified by the committee as "_1_".

The error rate of the committee on the training set was 18.8% – the same as the lowest error rate of the networks comprising the committee (and with the

threshold of 6 instead of 9 this rate was even lower: 18.1%). The committee was composed of 898 hidden neurons in total and was able to classify roughly 1000 patterns per second on AMD Duron 700 MHz.

## 5 Adaptability of the Classifier

In general, the simplest way of handling new data is to retrain the whole system once more, with the use of an augmented training set. This method is always valid, also for LTF-C. And for most neural networks, e.g., very popular Multi-Layer Perceptron (MLP), this method is the only possible (in the case of MLP the problem of local minima is the reason why any little modification of the training set requires retraining the network from the beginning).

However, LTF-C can also undergo continuous training, with the use of variable training set. The problem of local minima does not exist in LTF-C, thanks to the *locality* of transfer functions used in hidden neurons. This locality ensures that a hidden neuron has limited influence on the network response, so removing this neuron or modification of its weights influences only these patterns which lie in its reception field. In other words, every hidden neuron solves only small part of the classification problem. If this problem gets changed a little (with the change of the training set), only small part of neurons have to be modified, and not the whole network.

Most probably, in continuous training of LTF-C the training parameters should be constant. Thus, the coefficients $\tau_i^{(t)}$ (10) and $\tau_{\text{ins}}^{(t)}$ (16), which control the speed of the training, should be 1 all the time.

It should be mentioned, however, that the continuous training of LTF-C has not been tested, so far, and this issue needs further research.

## 6 Summary

This paper presented an application of LTF-C to recognition of active and non-active bank clients. LTF-C was chosen to solve this problem, because it had proved in other applications, such as handwritten digit recognition, credit risk assessment or breast cancer recognition, that it has very good accuracy, small size and short training time [1]. LTF-C is also fast and versatile – it is able to solve classification problems of different types.

After thorough data preprocessing and creating a committee of networks, LTF-C was able to achieve about 20% error rate in this problem.

Despite large amount of training data, this error rate was relatively high. This suggests that decision regions of both classes were overlapping, so in some parts of the input space it was not possible to discriminate between patterns from different classes. In other words, the attributes describing the samples did not hold enough information to perform correct classification.

Thus, in order to increase the accuracy of the classifier, it is necessary to add some new information to the input patterns, by adding entirely new attributes.

However, the usefulness of the system can also be increased in another way. Since the problem is of the financial domain, it is possible to estimate the cost of each type of misclassification. E.g., when the system classifies an active client as non-active, the bank bears the cost of unnecessary proactive action. This cost can be estimated. Similarly, when the system classifies a non-active client as active, the bank bears the cost of the loss of a client.

When we know the costs of misclassifications, we can create a so-called *cost matrix* [3], which can be utilized during the training. The use of the cost matrix has two advantages:

1. during the training the system pays more attention to important patterns – these which lead to high costs when misclassified,
2. the system performance can be expressed not in abstract terms of the error rate, but in the terms of profit or loss resulting from the use of the system. This indicator of the system usefulness is much better for the bank.

Although utilizing the cost matrix will not decrease the standard error rate (i.e., percentage of misclassifications), it will improve the profitability of the system.

## References


1. Wojnarski, M.: LTF-C – Neural Network for Solving Classification Problems. In: Wyrzykowski, R., et al. (eds.): Parallel Processing and Applied Mathematics. Lecture Notes in Computer Science, Vol. 2328. Springer-Verlag, Berlin Heidelberg New York (2002) 643–651
2. Fiesler, E., Beale, R. (ed.): Handbook of neural computation. Oxford University Press, Oxford (1997)
3. Michie, D., Spiegelhalter, D.J., Taylor, C.C.: Machine Learning, Neural and Statistical Classification. Elis Horwood, London (1994)
4. Merz, C.J.: Using correspondence analysis to combine classifiers. Machine Learning, Vol. 36, 33–58