Stochastic Weight Selection for Backpropagation Through Time Learning
(Theoretical study of stochastic neural networks learning)

Doctoral Dissertation Proposal

Study program: Artificial Intelligence
Study department: 9.2.8 artificial intelligence
Advisor: prof. Ing. Peter Sinčák, CSc.
Form of study: external

Košice, 29. June 2012
Ing. Juraj Koščák
# Contents

1 Introduction ............................................. 3  
1.1 Focus of the Dissertation ................................. 4

2 The State of the Art and Related Methods ................. 6  
2.1 Prediction Methods and Weather Prediction ................. 6  
2.1.1 Classical Methods of Weather Prediction ............... 8  
2.1.2 Auto-Regressive Moving Average Model (ARMA) ........ 11
2.2 Neural Networks for Weather Prediction .................. 12  
2.2.1 Neural Networks Introduction .......................... 13
2.2.2 Error Back-Propagation Method (BP) ................... 15
2.3 Recurrent Neural Networks (RCNN) ........................ 16  
2.3.1 Backpropagation Through Time (BPTT) ................. 17  
2.3.2 Batch and Online Learning BPTT ....................... 19  
2.3.3 Werbos Implementation of BPTT ....................... 21  
2.3.4 Shorted BPTT, Quick-Prop Through Time (QPTT) ....... 23  
2.3.5 Time Delay Neural Network (TDNN) .................... 24
2.3.6 Elman, Jordan and Bengio Neural Networks .......... 25  
2.4 Stochastic Methods ...................................... 28  
2.4.1 Echo State Networks (ESN) ............................ 29  
2.4.2 Random Weights Initialization/Reinitialization ....... 30  
2.4.3 Random Reordering of Patterns ....................... 31
2.4.4 Random Selection Update .............................. 32
2.4.5 Random Factor in Learning Constants ................ 32
2.5 Summary ................................................. 33

3 Proposed Algorithm Improvements .......................... 34
3.1 Prediction of Daily Temperature Profile Using Local Meteorological Data ................................. 35
3.1.1 Prediction System Implementation ..................... 35
3.1.2 Experiments and Results ............................. 39
3.2 Proposal of Stochastic Weight Update Method .......... 44
Chapter 1

Introduction

The mankind has always desired to know what will happen in the next days, months, future. The prediction, as a source of knowledge what will happen in the future, becomes very important. Ancient people wanted to know if the hunters of mammoths will succeed, what is waiting for them. Besides of the all other predictions, the weather prediction goes side by side with the evolution of civilization. It was always important to know the weather for agriculture, hunting, during building huge buildings. The weather has huge influence on the daily life of people. The prediction requires the knowledge of the wind, temperature, processes and interactions between events in nature. This knowledge is acquired during years and years of nature monitoring. Once we have such knowledge, it can be used and spread around.

Nowadays the information technologies become popular and are spread all around us. We do not consider them as something new, special, they become part of our daily life. We are in touch with them during our traveling, shopping, at work, even in our free time, or while doing some sports activities. The information technologies push our curiosity further and further. The curiosity and technologies are the main power of evolution for our civilization. Information technologies do bring the support for the prediction, too. New powerful computers that are able to calculate time-consuming simulations those computers with huge computation power are build regularly. The computation power cannot be used without software, algorithms that will control how the knowledge it has to be used. Artificial Intelligence is used to make computers less human dependent.

Once we put computers, software and artificial intelligence together, we are getting new dimensions of an opportunities to exploit the computation power and possibility to advance our inquisitiveness. We can do it smart, reuse some of the human intelligence and transfer it to the computers. In such combination there is high probability to achieve more accurate results, to build more autonomous machines.
1.1 Focus of the Dissertation

This work is inspired and motivated by the results of the Salvetti and Wilamowski experiment focused on the introduction of stochasticity into Backpropagation (BP) algorithm on experiments with the XOR problem [1]. The stochasticity can be embedded into different parts of the BP algorithm as it is more fully described in section 2.4. This exact implementation of the algorithm, in the original work [1], was discussed only briefly. This leads us to introduce and implement different types of BP algorithm modifications, that continuously involve more stochasticity on various parts of the BP algorithm.

This dissertation is focused on the modification of the Backpropagation Through Time (BPTT) algorithm and its implementation on the Recurrent Neural Network (RCNN). The modification of the algorithm lies in the learning phase, the execution phase of network is the same as in the original BPTT, specifically the weight selection and weight update processes. This modification consists of the stochasticity that involves all parts of the weight update and weight selection processes during their selection for adaptation and the direct adaptation of weights. The modification - stochasticity is used to choose which weight is going to be updated, which corresponding weights will be taken into consideration when calculating the new value of the current weight, and how many times the weight adaptation process is going to be repeated (see Figure 3.7). The stochasticity will also be involved in the selection of recurrent neurons, so that they will be part of the learning process too. This algorithm modification will be compared with a set of experiments with the classical BPTT algorithm implementation (see section 2.3.3). The experiments will be focused on the varied topology of the Neural Network, the learning parameter $\gamma$, the number of recurrent neurons, and the $\alpha$ momentum (see section 3.2.1). To have a better idea of the possible performance of the stochastic weight update and, what can be expected from this kind of modification, the experiment will be based on the classical algorithm for Feed Forward Neural Networks (FFNN)(see section 3.2.2) - the Backpropagation algorithm. Those experiments will be used to get the basic idea of the stochastic weight update performance and will be used as a starting point for RCNN using the BPTT learning algorithm. The FFNN will use a toy-task data circle-and-square, where the task is to correctly identify the points that belong to the circle or to the square. The toy-task data for RCNN will be a time-shifted and skewed signal (see Figure 3.14).

The goal of this work is to prove, that this stochastic modification is able to learn efficiently and the results are comparable to classical implementation. This stochasticity also brings a simpler implementation of the algorithm, than the classical one. This is especially useful on the RCNN.
The work has following research ambitions:

1. Provide theoretical background and analyze of the Recurrent Neural Networks, specifically focused on the time series Neural Networks and stochasticity in their learning process.

2. Propose stochastic modifications of Neural Network learning process.

3. Set up experiments and obtain experimental results with modified NN.

4. Evaluate realized experiments and analyze results.
Chapter 2

The State of the Art and Related Methods

This chapter is focused on a summary of variants of the classical method - the Backpropagation Through Time (BPTT) algorithm, its modifications and the usage of stochastic processes in backpropagation learning in the field of weather prediction.

2.1 Prediction Methods and Weather Prediction

A prediction (according to Howard 1998 [2]) is a statement of what will happen in the future which is based on observation from the past. Nowadays, most predictions are based on an experimental basis. In prediction, we distinguish between prediction and forecast. A prediction is an exact outcome that we expected, while a forecast covers a range of outcomes. It is an extrapolation of an unknown function. The prediction of time series is focused to create a model describing an unknown process based on available information and knowledge about the specific process. A newly created model is further used to simulate information in the feature [3].

The time series is created from the measuring or observation of natural, technical or economical processes, which are measured chronologically. The attribute of the value in time series is that each actual value depends on one or more values measured previously. Based on these dependencies hidden in the time series there is a chance to find a hidden function defined as:

\[ x(t + 1) = f(x(t), x(t - 1), x(t - 2), \ldots, x(t - (N + 1))) \]  

(2.1)

where \( x(t) \) represents state in time \( t \), function \( f \) represents dependence of state \( x(t + 1) \) from previous states of the depth \( N \). Based on collected data, the model
for a specific process is created.

The data set of the measured process should contain enough data, of which the
data set must be representative. In this case, the prediction is precise. This kind
of set in not always possible to get. Then the error of the prediction should be as
low as possible to keep valid predicted data, or a number of validation criteria [4].

Each prediction model has to be validated. There exist criteria based on which
the prediction based on the model can be validated. The criteria are:

- Accuracy of prediction,
- Adaptability of prediction,
- Robustness and reliability.

All the above mentioned criteria are important and consideration has to be taken
off them. The accuracy of prediction is the most important criteria. The accuracy
depends on a lot of factors. The highest impact have values, which are directly
related to the prediction. Also the time epoch has an influence on the prediction.
There are two types of models recently used for prediction:

- Linear model,
- Nonlinear model.

The linear model is an operation where the future values of a discrete-time
data are estimated as a linear function of its previous values:

\[ \hat{x}(t) = \sum_{i=1}^{p} a_i x(t - i), \]  

(2.2)

where \( \hat{x}(t) \) is the predicted data in time \( t \), \( x(t - i) \) are previously collected data, \( p \)
is the number of previously used data and \( a_i \) is the predictor coefficient. The \( e(t) \)
as a final error of prediction for time \( t \) is defined as:

\[ e(t) = x(t) - \hat{x}(t), \]  

(2.3)

where \( x(t) \) is the real measured data. It does not require huge computation power
and is easily used as a real time prediction.

On the other hand, the nonlinear model is more complex and requires huge
computational power. Until recently, it was not possible to use it in real time
computations. One of the well know models in this area are NNs. Later in this
work I will give a general overview of NNs, which can be found in [5].
2.1.1 Classical Methods of Weather Prediction

In the previous centuries, people were focused on signs observable in nature like an animal’s behavior and the movement of clouds. The arrival of computers and their improvement play a big role in weather prediction. It is their computing power which is used to simulate the mathematical models and dynamics of the weather. The methods of classical weather prediction can have a lot of variations. The most commonly used are:

1. Persistence Method,
2. Climatology Method,
3. Analog Method,
4. Trends Method,
5. Numerical Method,

**Persistence Method**

There are several different methods that can be used to create a forecast. The method a forecaster chooses depends upon the experience of the forecaster, the amount of information available to the forecaster, the level of difficulty that the forecast situation presents, and the degree of accuracy or confidence needed in the forecast.

The first of these methods is the Persistence Method (according to the University of Illinois [6]). This is the simplest way of producing a forecast. The persistence method assumes that the conditions at the time of the forecast will not change. For example, if it is sunny and 31°C Celsius today, the persistence method predicts that it will be sunny and 31°C Celsius tomorrow. If two inches of rain fell today, the persistence method would predict two inches of rain for tomorrow.

The persistence method works well when weather patterns change very little and features on the weather maps move very slowly. It also works well in places like southern California, where summertime weather conditions vary little from day to day. However, if weather conditions change significantly from day to day, the persistence method usually breaks down and is not the best forecasting method to use.

It may also appear that the persistence method would work only for shorter-term forecasts (e.g. a forecast for a day or two), but actually one of the most useful roles of the persistence forecast is predicting long range weather conditions.
or making climate forecasts. For example, it is often the case that one hot and dry month will be followed by another hot and dry month. So, making persistence forecasts for monthly and seasonal weather conditions can have some skill. Some of the other forecasting methods, such as numerical weather precipitation, lose all their skill for forecasts longer than 10 days. This makes persistence a "hard to beat" method for forecasting longer time periods.

Climatology Method

The Climatology Method (according to the University of Illinois [6]) is another simple way of producing a forecast. This method involves averaging weather statistics accumulated over many years to make a forecast. For example, if you were using the climatology method to predict the weather for Košice City on the 4th of July, you would go through all the weather data that has been recorded for every July 4th and take an average. If you were making a forecast for temperature and precipitation, then you would use this recorded weather data to compute the averages for temperature and precipitation and not use current conditions.

If these averages were 30° Celsius with 0.45 centimeters of rain, then the weather forecast for Košice City on July 4th, using the climatology method, would call for a high temperature of 30° Celsius with 0.45 centimeters of rain. The climatology method only works well when the weather pattern is similar to that expected for the chosen time of year. If the pattern is quite unusual for the given time of year, the climatology method will often fail.
Analog Method

The Analog Method (as described by the University of Illinois [6]) is a slightly more complicated method of producing a forecast. It involves examining today’s forecast scenario and remembering a day in the past when the weather scenario looked very similar (an analog). The forecaster would predict that the weather in this forecast will behave the same as it did in the past.

For example, suppose today is very warm, but a cold front is approaching your area. You remember similar weather conditions last week: also a warm day with a cold front approaching. You also remember how heavy thunderstorms developed in the afternoon as the cold front pushed through the area. Therefore, using the analog method, you would predict that this cold front will also produce thunderstorms in the afternoon.

The analog method is difficult to use because it is virtually impossible to find a perfect analog. Various weather features rarely align themselves in the same locations they were at the previous time. Even small differences between the current time and the analog can lead to very different results. However, as time passes and more weather data is archived, the chances of finding a ”good match” analog for the current weather situation should improve, and so should analog forecasts.

Trends Method

The Trends Method (as described in ECMWF [7]) uses the speed and direction of front movements, high and low pressure centers, areas of clouds, and precipitation to make predictions. Using this method, the weather in other areas can be tracked to determine when they will arrive and what conditions they will deliver. It means, that if the storm is 300 kilometers west of the current position that requires a forecast, and it moves 75 kilometers per day, it will arrives in 4 days:

\[ d(t) = \frac{s(t)}{v(t)}, \]

\[ (2.4) \]

where \( d \) is a number of days ahead in which the storm arrives, \( s \) the distance of the storm from the current area, \( v \) is the speed of the storm and \( t \) is time when the computation was done. This method uses the calculation of mathematical equations that are used for the short-term. However, should the variables of the equation change (i.e, the system speeds up, the wind speed changes, the direction changes) the Trends Method becomes much less accurate.

Numerical Method

The Numerical Weather Method (as described by the ECMRWF [8]) is the most predictable and requires the largest computer power of the mentioned methods.
The forecast model is generated by a complex set of calculations based on multiple atmospheric conditions. While more advanced than the other methods, the Numerical Weather Method is still flawed because it lacks data from conditions over the ocean and at high mountain elevations which are not tracked.

**Time Series Method**

This method is the most complex one. It uses historical data for weather prediction. The Time Series is a sequence of data depending on time $t$, that were measured at uniform time intervals $x(0), x(1), ..., x(t)$. Part of this time series are methods like the Autoregressive Moving Average (ARMA) and the Autoregressive Integrated Moving Average (ARIMA) that were mentioned earlier in this work. The aim of the Time Series analysis is to describe the measured data and make predictions based on them.

### 2.1.2 Auto-Regressive Moving Average Model (ARMA)

ARMA (as described in Box, G. and Jenkins M., G. and C. Reinsel, Gregory [9]) - Auto-Regressive Moving Average is a combination of the Auto Regressive (AR) Model and the Moving Average (MA) Model approach. This model is used to describe statistics, signal process models which contain a time series inside. It provides an understanding of the time series in data, predicting feature values in the series. ARMA is used to model data with no trend and data containing some unobserved shocks combined with system behavior. For example, stock price can be shocked by fundamental information combined with exhibiting technical trending and mean-reversion effects due to market participants.

The **Auto-Regressive** ($AR(p)$) model of ordered $p$ states is described by the equation:

$$
\hat{x}(t) = \sum_{i=1}^{p} a_i x(t-i) + \epsilon(t),
$$

(2.5)

where $\epsilon_t$ is white noise (random signal containing equal power within a fixed bandwidth at any center frequency), $p$ is number of ordered states of model from previous time $t$ and $a_1, ..., a_p$ are model parameters.

The **Moving Average** ($MA(q)$) model description by number of ordered previous states $q$:

$$
\hat{x}(t) = \mu + \epsilon(t) + \sum_{i=1}^{q} b_i \epsilon(t-i),
$$

(2.6)

where $b_1, ..., b_q$ are model parameters, $\mu$ is expected value of $x(t)$ usually 0 and $\epsilon(t), ..., \epsilon(t-i)$ are white noise.
CHAPTER 2. THE STATE OF THE ART AND RELATED METHODS

The combination of $AR$ and $MA$ signals gives the ARMA. The mathematical description is represented by the equation:

$$x(t) = \epsilon(t) + \sum_{i=1}^{p} a_i x(t-i) + \sum_{i=1}^{q} b_i \epsilon(t-i).$$

(2.7)

ARMA models in general can, after choosing $p$ and $q$, be fitted by MSE to find the values of the parameters which minimize the error term. It is generally considered good practice to find the smallest values of $p$ and $q$ which provide an acceptable fit to the data.

Finding the appropriate values of $p$ and $q$ in the $ARMA(p,q)$ model can be acquired by plotting the partial autocorrelation functions for an estimate of $p$, and the same way using the autocorrelation functions for an estimate of $q$.

Adaptation of ARMA model was a basis for development of Backpropagation algorithm by Werbes [20].

2.2 Neural Networks for Weather Prediction

There are several different methods that can be used to create a forecast. Neural networks provide a methodology for extracting patterns from noisy data. They have been applied to a wide variety of problems in a meteorological context, including cloud classification (Bankert 1994) [10] and tornado warnings (Marzban, Stumpf 1996) [11]. The advantage and disadvantage of neural networks, in comparison to the other statistical techniques for pattern extraction, is discussed in (Marzban, Stumpf 1996) [11]. More details about the construction of neural networks can be found in (Marzban, Stumpf 1996) [11].

Doppler radar was used as a source of data for prediction using NN. The Meso-cyclone Detection Algorithm (MDA) from the Severe Storms Laboratory (NSSL) was designed to search for patterns in Doppler radar data, which are associated with rotating updrafts in severe thunderstorms. The storm-scale circulations are precursors to tornadoes. The NN was used to identify such circulations that were detected by NSSL using MDA, which later can yield to tornadoes. The input was represented by 23 variables, which characterize the type of circulations. The output of the NN is the boolean value, which says, based on the circulations, the tornado exists/not-exists. The exist/not-exist from those circulations were based on ground observation. Based on these experiments, the accuracy of NN prediction was comparable to the classic techniques.
2.2.1 Neural Networks Introduction

A neural network (Figure 2.2) is a mathematical model that is inspired by biological neural networks and tries to simulate them. It consists of interconnected units - neurons, which are the computation units of a neural network. NNs are part of Artificial Intelligence. The knowledge is stored in connections between neurons which are called synaptic weights (weights), simplification of biological dendrites and axons. NN is a universal approximator of relations stored inside data - a nonlinear statistical data modeling approximator, is able to learn and adapt its structure based on internal/external information that is propagated through the NN during the learning phase. It is relatively easy to use in a wide area of technical and nontechnical areas without further theoretical knowledge for most NNs. There is a number of NNs that require knowledge to implement them and use the correct set of initialization parameters. By definition [5], it is a massively parallel distributed processor, that is able to store experimental knowledge and is able to make it available for further use. It is able to:

- retrieve knowledge from propagating data in the NN during the learning phase, where weights $w$ values are not fixed

\[ \frac{\partial w}{\partial t} \neq 0, \]

(2.8)

- reuse the gained information stored inside of its fixed weights values $w$

\[ \frac{\partial w}{\partial t} = 0. \]

(2.9)

Figure 2.2: Artificial feedforward neural network
CHAPTER 2. THE STATE OF THE ART AND RELATED METHODS

The learning phase is a process of weights modification in order to achieve the required objective of the learning. It is a process of searching for a minimum of mean square error (MSE) for a concrete data set. During the learning phase, the knowledge of the NN represented by its weights, represents a nonlinear mapping $M(X; W)$, which is applied to the area in attribute space $\mathcal{X}$. For $X \in \mathcal{X}$, $M(X; W) = Y \in \mathcal{Y}$, where $W$ is the actual vector of weights [12]. Using an algorithm based on minimizing of MSE can cause two NNs with the same MSE to differ in internal representation [13] and [14]. At first glance, it looks like the NN is a black box with no way to see how it works inside.

An NN can be divided into two parts from the internal connection point of view between neurons:

- Feed Forward Neural Networks (FFNN),
- Recurrent Neural Networks (RCNN).

Neurons in an FFNN (see Figure 2.2) are connected from the front to the end and synapses between neurons are in one direction. The signal spreads only one way, from the input neurons to the output neurons. FFNN are easier to implement and they have no problem with their internal stability.

Neurons in an RCNN are connected as in an FFNN and additionally connected, or connected with neurons in the previous layer (see Figure 2.3).

The learning of an NN can be described by an $(n + 1)$ dimensional surface called the error space. The $n$ represents the number of weights $w_1, \ldots, w_n$ (dimensions) presented in the NN. This means that each weight and threshold represents one of the dimensions. The last dimension is the error of the NN in the current state with the current values of weights. The training algorithm of an NN iteratively seeks the global minimum of this surface. It often happens that the minimum is local and the NN has a problem getting out of this minimum. The gradient of the error
surface is calculated at the current point and is used to make a downhill move.
More details about FFNNs can be found in [5].

2.2.2 Error Back-Propagation Method (BP)

The method of error back-propagation (BP) belongs to the group of algorithms based on supervised learning. Supervised learning is used to learn feed forward neural networks (FFNN). It is widely used for FFNN. The BP was designed in 1986 [15], and a similar learning algorithm was designed in 1974 [16].

The target of BP is to achieve the state of weights, where the error between required output and output produced by neural networks was minimal corresponding to the training set. In the learning phase of an NN, where patterns are presented one by one so called online learning, the following equation is used to calculate the error of the output neurons for presented pattern:

$$ E = \frac{1}{2} \sum_{i=1}^{N} (d_i - o_i)^2, $$

(2.10)

where $E$ is the error of the presented pattern to the NN, $N$ represents the number of output neurons, $d_i$ the desired output of the NN for $i$’s neuron, and $o_i$ the real output of $i$’s neuron.

The method of rapid gradient grow is used to search for the minimum of error function $E$. The $E$ is general error of an iteration during learning process of an NN. This means that the weights are adapted to ensure the rapid drop of the $E$ function. The weights are changed fixedly to the negative gradient $-\frac{dE}{dw}$. The weight changes are:

$$ \Delta w_{ij} = -\gamma \delta_i x_j, $$

(2.11)

where $\gamma$ is the learning parameter mostly chosen from the interval $< 0, 1 >$ and $\delta_i$ is the error signal produced by $i$’s neuron:

$$ \delta_i = \frac{\partial E}{\partial in_i}, $$

(2.12)

and $x$ is the output of $j$’s incoming neuron into $i$’s current calculating neuron:

$$ x_j = \frac{\partial in_i}{\partial w_{ij}}, $$

(2.13)

calculating the error of the output neurons, this error is back propagated to the input neurons. For each neuron in the hidden layer error signal is calculated, which is influenced by the signal from the previous neurons connected with them:

$$ \delta_i = \left( \sum_{h=0}^{Nh} \delta_h w_{hi} \right) (f_i)', $$

(2.14)
where $f'_i = \frac{\partial f_i}{\partial \theta_i}$ is a derivation of $i$’s neuron of $j$’s layer, $h$ is the neuron of previous $j$’s layer and $Nh$ is the number of neurons from $j$’s layer. A complex description of the whole BP algorithm can be found in [3].

The NN learning can be explained using concept of error space. The error space is $(n + 1)$ dimensional space. The $n$ represents the number of weights $w_1, \ldots, w_n$ (dimensions) inside the NN. This means, that each weight and threshold represents one of the dimension. The $(n + 1)$’s dimension is the error of the NN in the current state with the current values of weights. The training algorithm of an NN iteratively seeks the global minimum of this surface. It often happens that the minimum is local and the NN has a problem getting out of this minimum. The gradient of the error surface is calculated at the current point and is used to make a downhill move.

2.3 Recurrent Neural Networks (RCNN)

Recurrent neural networks (RCNN) [5] (see Figure 2.3) are NN where synapses are connected in different directions. Each neuron is at the same time an input and an output. This kind of neuron is also known as a dual neuron. The basic conentional view of using RCNN is described in [17]. The reasons for using RCNN instead of FFNN are:

- if FFNN does not produce sufficient results
- if FFNN is not effective enough based on requirements for accuracy to solve the current problem.

RCNNs are in general more complex dynamic system than FFNNs. The problem of neurodynamics is important for global stability and convergence during the training of an FFNN. The learning approach is the same as by FFNN and can be divided into two groups:

- Supervised learning of RCNN - BP for RCNN, Elman RCNN ...
- Unsupervised learning of RCNN - ART method of learning ...

**Supervised learning** is a process where patterns to which we already have an expected outputs that an RCNN should provide are presented as an input to an NN. The expected output can be identical to the input mostly in the case of memory simulation.

**Unsupervised learning** A group of methods that represents unsupervised learning on RCNN is the Adaptive Resonance Theory (ART) approach, also called Cluster discovery networks. In generally can be characterized as a method of
CHAPTER 2. THE STATE OF THE ART AND RELATED METHODS

clustering. The problem is, that till now, there is no proof of global stability using competitive learning of RCNN, which can cause problems with network stability. One way to solve it is a continuously decreasing learning parameter $\gamma$ to 0. In this case, there will be global stability for sure, but on the other hand, the NN will not be so adaptive - it loses its plasticity. It will not be able to react to new inputs coming to the NN. This problem was identified by Stephan Grossberg [18] and was called the stability-plasticity dilemma. The resonance, which is a connection between the input and the created cluster, tries to solve this problem.

As a part of RCNN there is a sub-group of partly recurrent NNs. In this kind of NN there is a special group of neurons, which do not sufficiently satisfy the definition of recurrent neurons. This partly of RCNN was designed by Elman and Jordan. They are mostly used for sequential processing of a sample as one unit [19].

2.3.1 Backpropagation Through Time (BPTT)

Error Back-Propagation Through Time (BPTT) is a learning algorithm used for RCNN derived from classic BP used on FFNN. The advantage of the BPTT is that, besides full topology, the implementation is relatively easy compared to other learning algorithms. Compared to Hebb rule, BPTT is based on the BP which is more powerful. On the other hand, BPTT has a limited depth of recursion defined by parameter $T$. BPTT is is the most practical implementation of recurrent BP and it is more efficient learning than methods based on Hebb rule. BPTT can be applied to neural networks and to econometrics. The network adaptation is done in discreet time steps. BPTT on RCNN is just an extension of classic BP on FFNN by calculating the error influenced by error from the previous time series steps of RCNN. The error influencing NN from the previous time steps give rise to the problem of stability and convergence for NN, while the current state of NN is influenced by the step from the past of the same NN. More information about NN stability and convergence can be found in the literature [6]. One of the reasons for transferring RCNN to a similar FFNN is to avoid problems with NN stability.

The core of the BPTT algorithm is that the RCNN (the neurons of the RCNN don’t contain a connection to the previous layers, just a connection to neurons in the next layer) is transformed - "unfolded" to the FFNN containing more layers, where the classic BP can be applied with small changes. By this unfolding of the RCNN, each newly created layer of FFNN represents one recursion in time of the RCNN labeled by $T$ - the depth of the unfolding of the RCNN (the number of layers - 1 in the FNN represents the number of past time - recursions represented by the original RCNN). Each neuron in such a hidden layer keeps its activation in time $t$ and in time $t + 1$ serves this activation of the input of all the neurons in the hidden layer, also for itself. This is done in the next step. The whole hidden layer
sends in time \( t \) its output to the output layer at the same time as it sends its own output to itself in the next time step \( t + 1 \). If we visualize some part of the RCNN using the “unfolding” mechanism, we get a multilayer FFNN as it is displayed in Fig. 2.4. This figure represents the unfolded RCNN with recurrence depth 3, 3 inputs followed one by one (the current input \( t \), inputs from the previous time steps \( t = i \)). It is really important to release the dependence between each single weight in such an unfolded NN. The weights \( U \) between hidden layers are the same for all inter-layer weights, weights between the input layer and the hidden layer \( V \), and weights between the hidden layer and the output layer are the same for each layer. The modification of classic BP for such an unfolded NN is based on small changes during weights adaptation. By propagating an error signal through all layers and through all time layers calculated all weight differences \( \frac{dE}{dw_{ij}} \) are calculated.

These weight differences are not applied immediately to the current calculating of weights, but they are used as a contribution to the complete change of weights. The weight adaptation is done in time, when we are back at the starting time, where the weights are adapted using a complete contribution of each weight from a particular time series:

\[
\Delta w_{ij}(t) = -\gamma \sum_{k=0}^{T} \frac{\partial E(t)}{\partial w_{ij}(t-k)},
\]

(2.15)

where \( T \) represents depth of unfolding of the RCNN - unfolded NN to the
history, $\gamma$ is the gradient of learning (speed of learning) and $w_{ij}(t)$ is the weight in time $t$.

This kind of RCNN unfolding requires a lot of memory for NN, that contains a lot of neurons in one layer. If the depth parameter $T$ is too large, a gradient effect can disappear inside of the RCNN. The back-propagated error spreading through all the layers can decrease to such a small value, that it can be taken as 0 - the error just disappears. The unstable state of the RCNN can be caused by choosing an ambiguities delta [8], which causes at change in the convergence of the RCNN. To make the learning of RCNN easier, there are a lot of different modifications of the BPTT algorithm.

By modifying the BPTT algorithm, we can achieve a more effective algorithm, decrease the time consuming side of the algorithm, decrease the memory requirements for the unfolding of RCNNs. The modification can take part of the algorithm, modifying the structure of the NN which can improve the storing of the history inside the NN, accumulating values from the history using different structure of NN topology. The modification of some part of the NN closely connected to the problem, that is being solved by the NN. This chapter provides an introducing to the commonly used modifications of the BPTT algorithm.

2.3.2 Batch and Online Learning BPTT

Batch learning is a type of learning where the data are given to the input of the NN in batches, and after each such batch, the weights of the NN are adapted. The NN during processing of batch data, containing $T$ inputs, is unfolded with depth $T$ - history. After this, the error is back propagated through the NN and are applied to the calculated changes of weights. This kind of learning is used by the inference of grammar. One batch is represented by one word, which consists of sets of symbols. After the end of such a sequence, an output signal is required. The words (batches) do not depend on each other. The NN is reset after each batch to the "reset state", which depends on currently solving task. This reset is done in the learning phase of the NN and also in the life of the NN. The NN works epoch by epoch.

The batch learning can be also used in continual input sequences, where the input data do not consist of separate sections. In this case, the NN is not reset after each batch. This kind of learning approach can be used for prediction tasks and generative tasks. In this type of task, the error information is available after each presented symbol. The error is not put just at the end of each batch of symbols. Based on the unfolded NN, described in Figure 2.4, it is possible to detect that the error can be applied at the end (in the output layer), but it also can be applied in any other time steps (output layer in $t \neq 0$). Using such an approach to the learning of the NN, two different usages of batch learning are created. The followed equation
is used to calculate the error signal of the output layer only at the end \( t = T \):

\[
\delta_i(T) = (d_k(T) - o_k(T))(f_k(T))'.
\]  
(2.16)

To compute the error signal for all hidden layers \( K \) at the end \( t = T \) is used equation:

\[
\delta_j(T) = \left( \sum_{k=1}^{K} \delta_k(T)w_{kj}(T) \right) (f_j(T))',
\]  
(2.17)

and for the rest of the \( J \) hidden layers where \( t \neq T \):

\[
\delta_j(t) = \left( \sum_{l=1}^{J} \delta_l(t+1)u_{lj}(t+1) \right) (f_j(t))', \ t_0 < t < T.
\]  
(2.18)

If the error is introduced into the NN in each time step, the equation 2.18 is changed to the equation 2.19, where the error signal of the output layer is added to the current time step:

\[
\delta_j^t = \left( \left( \sum_{k=1}^{K} \delta_k(t)w_{kj}(t) \right) + \left( \sum_{l=1}^{J} \delta_l(t+1)u_{lj}(t+1) \right) \right) (f_j(T))', \ t_0 < t < T.
\]  
(2.19)

For both cases, the equation for weight change between the hidden and the input layer are as described bellow. The weights \( v_{ji} \) between input and hidden layer:

\[
\Delta v_{ji} = \alpha \left( \sum_{t=t_0}^{T} (T)\Delta v_{ji}(t) \right) = \alpha \left( \sum_{t=t_0}^{T} (T)\delta y_j(t)x_i(t) \right),
\]  
(2.20)

the weights between the hidden layers \( u_{jl} \):

\[
\Delta u_{jl} = \alpha \left( \sum_{t=t_0}^{T} \Delta u_{jl}(t) \right) = \alpha \left( \sum_{t=t_0}^{T} \delta y_j(t)y_l(t-1) \right).
\]  
(2.21)

The equation for the change of weights \( w_{kj} \) between the output layer and the hidden layer, if the error signal taken in the consideration just at the end \( t = T \) step, looks like:

\[
\Delta w_{kj}(t) = \alpha \delta_k(T)y_j(T).
\]  
(2.22)

If the error is inserted into each time step, the total change of weights between the hidden and the output layer and the hidden and the input layer is calculating using equations 2.20 and 2.21. The summation of the increment in concrete time steps:
Incremental Learning of BPTT

Incremental learning of NN means continuously processing inputs one by one as they follow by themselves. The back propagation of error and weights modification is done after each individual input data. This approach can be used in the epoch mode of the NN, but from basic principle (weight modification after each input), it is mainly used in the real time learning of the NN. Due to this the method is also called BPTT in real time (Real-Time BPTT).

It is also called the incremental method, because it continuously unfolds the NN while it processes input data. This algorithm can be better explained using Fig. 2.4. After the NN processes the first input, the NN is in the state time $t = 1$, classic FFNN with one hidden layer. The error is back propagated and the weights are adapted. After the second input is processed, the NN is unfolded again, as the NN in figure Fig. 2.4 for the time $t = 2$ - two hidden layers. In this case, the error is inserted just into the last output layer. In this case the output layer is the layer in time $t = 2$. The error is back propagated and the weights are updated. Then the 3rd input is processed (has 3 hidden layers and it contains information from the 1st, 2nd and 3rd inputs) the same way as the 1st and 2nd inputs while all the inputs aren’t processed. This algorithm is also called BPTT, because it is able to store unlimited previous states of the NN. Due to its large memory requirements it is a more theoretical than practical algorithm.

2.3.3 Werbos Implementation of BPTT

The original Werbos [20] implementation in Figure 2.5 represents batch learning of the NN, after each epoch (set of data) consisting of some amount of values, all weights are adapted. This kind of learning for the NN is suitable in the cases of generalizing the training set (cluster), recognition applications, dynamic modeling, sensitivity analysis, control system over time. The text recognition belongs to the area of generalization application, where one letter A can be represented by slightly different variations of the same letter, but still it represents the letter A. This kind of learning is not really useful in cases where the learning set is represented by a different set of data. The generalization feature in such a case is not required. The unfolding of RCNN to FFNN is not needed to be implemented. The RCNN unfolding represents the z-axis of the cubic (time dimension - history of NN), while the x-axis and y-axis represent the weight indexing of the NN.
N is the number of neurons in all layers, n represents the number of output neurons which starts from \( N + 1 \) till \( N + n \). The \( m \) represents the number of input neurons.

\( N \) is the number of neurons in all layers, \( n \) represents the number of output neurons which starts from \( N + 1 \) till \( N + n \), also \( N + n \) represents the number of all neurons. The \( m \) represents the number of input neurons, where inputs to the network are described as \( x_1 \ldots x_m \), they represent the input vector \( x \). All neurons inside are connected with the rest of the neurons inside the NN. The NN is fully connected. Even the output neurons, which generate the vector output \( y_n \), takes input from other output neurons such as one \( \hat{y}_{n-1} \). To simplify such full connecton between neurons, some weights \( w_{ij} \) can be set to zero (\( w_{ij} = 0 \)). Those weights are dropped from the calculation process during the learning of the NN. The Werbos approach tries to unfold the RCNN into a simple FFNN and tries to apply the classic BP learning algorithm for FFNN. All connections have to be considered in the learning process. The input in the neurons is represented by the summation of the outputs through all times \( T \) (\( T \) - depth of the unfolded NN and past representations inside the NN):

\[
in_i(t) = \sum_{j=1}^{i-1} w_{ij} x_j(t) + \sum_{j=1}^{N+n} w_{ij} x_j(t - 1) + \ldots + \sum_{j=1}^{N+n} w_{ij} x_j(t - T),
\]

(2.24)

where \( m < i \leq N + n \), \( m \) is number of input and hidden neurons, \( 1 \leq m \leq N \), \( N \) is number of neurons inside of input and hidden layer and \( n \) is number of output neurons (see Figure 2.5). To calculate the derivation of \( \partial w_{ij} \) equation of calculated derivation of \( \partial x_i(t) \) is used:

\[
\frac{\partial E}{\partial x_i(t)} = \frac{\partial E}{\partial \hat{y}_{i-N}(t)} + \sum_{j=i+1}^{N+n} w_{ij} \frac{\partial E}{\partial \hat{x}_i(t)} + \sum_{k=1}^{T} \sum_{j=m+1}^{N+n} w_{ji}^{k'} \frac{\partial E}{\partial x_i(t-k)},
\]

(2.25)

This equation makes it not possible to calculate \( \frac{\partial E}{\partial x_i(t)} \) until all \( \frac{\partial E}{\partial x_i(t+1)} \ldots \frac{\partial E}{\partial x_i(t+T)} \).
are already known. Therefore $\frac{\partial E}{\partial x}$ is calculated for the time $t = T$ and is calculated backwards to $t = 1$. To adapt weights $\frac{\partial E}{\partial w_{ji}}$ is used:

$$
\frac{\partial E}{\partial w_{ji}}^{k'} = \sum_{t=1}^{T} \frac{\partial E}{\partial x_i(t+k')} * x_j(t),
$$

(2.26)

where $k'$ is the weight calculated for the concrete time in the past ($0 < k \leq T$).

The modifications mentioned above give an overview of the main important part of the Werbos implementation of the BPTT algorithm. More details and pseudocode can be found in his book [20].

### 2.3.4 Shorted BPTT, Quick-Prop Through Time (QPTT)

Shorted BPTT is a genuine approximation of the BPTT($\infty$) method. By this approach, the training is upraised by limiting a back propagating of error in BPTT($\infty$) to the specified number of steps in past. By shortening of the back propagated error to the $h$ level, the error is the not back propagated from the current time $t = \tau$ till time $t = 1$ (through the hidden layers), but just till the time $t = \tau + 1 - h$ (maximum through $h$ last hidden layers). Shortened BPTT is also marked as BPTT(h), because it requires just $h$ steps of the history from the previous steps of time. This number of time steps (number of the hidden layers), through which the error information is propagated, is also called the time window.

During the gradient calculating, the whole history of inputs is not taken into consideration, just the inputs in the time window. The NN during the modeling of connections between individual inputs ignores the time dependence overhanging the size of the history window. This ignoring of the time context overhanging the $h$ time steps comes out of the core of the imperfect gradient method. The elements of current sequence have higher influence on the result of the gradient, then the elements further from the time window. This kind of phenomena is also called problem of information blocking [21]. It means, that using BPTT has not a big meaning to unfold the NN into high history deep, because the influence of the back propagated error is vanishing quadratically with the coming time steps (“vanishing gradient effect”). The vanishing gradient effect is caused by the storing training of data inside the NN and losing approximation ability.

Besides the method BPTT(h), raising from the incremental way of training, there exists also the shortened BPTT combining BPTT(h) with the batched approach. In this method, the back propagated error is restricted to the $h$ previous time steps. The difference is in the back propagated error and in the weight adaptation of the network, that treats $h$ next inputs and then the error is back propagated and the weights are adapted. The main difference is, that the weights adaptation is not done after each treated input, but always after $h$ inputs (after a batch with the
size of \( h \) This method is also known as BPTT(\( h; h' \)). Then the BPTT(\( h \)) can be identified as BPTT(\( h; 1 \)) and the continual batched processing (without network reset) with batch size \( h \) is BPTT(\( h; h \)). The processing complexity is lower, if the ratio \( h/h' \) is lower. With the decreasing of this ratio comes also a decreased accuracy of approximation for the real gradient. It is important to choose an accurate ratio of \( h/h' \).

**Quick-Prop Through Time (QPTT)**

Another modification of BPTT is called Quickprop through time [22]. To speed up the learning algorithm of the NN it uses the information about a curved surface in case that the error surface is a parabola. This information about a curved error surface can be gained from the second derivation of the error function. Quick-Prop divides the error surface into the quadratics and tries to jump to the parabola’s minimum in one step. The error derivation is calculated by each weight. Before the gradient is calculated using standard BP rules, the change of the weights is calculated using the equation:

\[
\Delta w_{ij}(t+1) = \frac{S(t+1)}{S(t) - S(t+1)} \Delta w_{ij}(t),
\]

(2.27)

where \( S(t+1) \) is the partial derivation of error by \( w_{ij} \), \( S(t) \) is the last calculated derivation for the concrete weight \( w \). This algorithm modification is useful for parity calculating or decoding problems. Using this method for the other problems, this can case a stagnation of the neural network [23].

**2.3.5 Time Delay Neural Network (TDNN)**

Simplification of the BPTT algorithm can also be done by modifying the network topology. The easiest modification of the network to capture the time context by FFNN is to use the time delay window. A network with such a modification is called a Time delay neural network (TDNN). This kind of network, displayed in Figure 2.6, has the information from the current time \( t \) and also information from the past \( t - 1, t - 2, ..., t - D \), where \( D \) is the size of the time delay window. In this case, is it not required to modify the learning algorithm. It is possible to use the classic back-propagation algorithm to learn such a neural network. The disadvantage of this method is the right size of the window, which in some cases can-not be large enough to capture the time context contained inside of the data. This kind of architecture is not good enough to be used for generation sequence by automatic machines (automatic machines persist in some states longer than the size of the time delay window).
2.3.6 Elman, Jordan and Bengio Neural Networks

Elman Neural Network

This method is focused on simplifying of the BPTT, designed and implemented by Elman in 1990 [25] (also known as the Simple Recurrent Network - SRN). The network uses classical neurons and also so-called context neurons (context layer), which represents an extra input from the previous steps of the hidden layer. The context neurons allows recurrence in the network (creates memory) with one step in the past. The input to the hidden layer is the input layer plus same hidden layer from the previous step, from the past. In this case is the unfolding of the neural network is not applied like in the previously mentioned implementations. The network consists of an FFNN that contains additional inputs from the context neurons which are recurrent to them-selves and create an easy RCNN, where the BPTT algorithm for weight adaptation can be used.

The experiments of the activation function proved, that the best function is the sigmoid function with input values from the interval \( < 0.2, 0.8 > \) instead of \( < 0, 1 > \), so weights will not have the infinity values [26].

It seems that the Elman network is similar to the Jordan network, but those networks are totally different. The differences between these networks are:

- the hidden layer is the context layers instead of the output layer,
- the context neurons are not recursive.

Hidden neurons are interconnected with context layer using weights which are set to value 1. The learning of such a network consists of:
Figure 2.7: Elman neural network (picture based on Elman 1990 [25])

1. initialization of context neurons by value 0 at the beginning in time $t = 0$,
2. pattern $x^t$ is given to the input and the signal is propagated,
3. applying BP rules to the whole network,
4. move to the next step $t = t + 1$, following step number 2.

The value of the weights that connect the hidden layer with the context layer equal the value in the previous step $t - 1$.

In computing the error signal for the Elman network using BPTT, the equation (2.14) is replaced by the corrected equation regarding time steps to the past:

$$
\delta_j(t - 1) = \sum_h \delta_h(t) u_{hj} f'(y_j(t - 1)).
$$

(2.28)

The sequential input is going to be processed by a clock which regulates presentation of the input to the network. Processing would then consist of the next steps. At time $t$, the input units receive the first input in the sequence. Each input might be a single scalar value or a vector, depending on the nature of the problem. The context units are initially set to 0.5.2. Both the input units and the context units activate the hidden units; and then the hidden units feed forward to activate the output units. The hidden units also feed back to activate the context units. This constitutes the forward activation. Depending on the task, there may or may not be a learning phase in this time cycle. If so, the output is compared with a teacher input and backpropagation of error is used to incrementally adjust connection strengths. Recurrent connections are fixed at 1.0 and are not subject to adjustment. At the next time step $t + 1$ the above sequence is repeated. This
Figure 2.8: Jordan NN the outputs are connected with state neurons, which are recurrent (picture based on Jordan 1986 [27]). The number of context neurons is the same as the number of output neurons.

At time the context units contain values which are exactly the hidden unit values at time $t$. These context units thus provide the network with memory [25]. The time is represented in the context units, that remember the previous steps. They store information from the internal state as well as the external state.

**Jordan Neural Network**

The Jordan network is another modification of the BPTT method. This network belongs to the first RCNN, which were publicized in 1986 [27]. In this NN, the signals from the output layer are given to the input layer using special inputs called state units (epoch units). The number of those neurons is the same number of output neurons. Those state units contain connections to them-selves. Context neurons contain an associated parameter $m$, which is in general $0 < m \leq +1$. It represents the sensitivity of the context neurons in keeping this information. The memory of the NN is created by the connection between the output and input of the NN and the connecting state neurons to them-selves. The NN complexity is lower than the regular layered RCNN. To learn such a network due to this lower complexity it is possible to use one of the modified BPTT algorithms. Figure 2.8 represents an example of a Jordan NN.

In case of the usage NN for prediction, there is just one output neuron, which represents the predicting value. Then it has only one context neuron and its activation in time $t$ is represented by:

$$x_c(t) = m \ast x_c(t - 1) + x(t - 1),$$  \hspace{1cm} (2.29)
where \( x_c \) is the activation of the context neuron, an associated parameter \( m \) which is in generally \( 0 < m \leq +1 \), \( x(t-1) \) is out of network from the previous step \( t - 1 \). The activation of the rest of the neurons is calculated as in the classic MLP, where the input is a combination of external context input and the input of context activations.

\[
f_{\omega}(t) = (x(t), ..., x(t-d), c(t))
\]  

(2.30)

**Bengio Neural Network**

This modification of the BPTT algorithm is a combination of the Elman NN and the Jordan NN. As it was designed by Bengio [21], it is possible to have an extra context layer for the different layers of the original multilayer NN as it is described in Figure 2.9. As a learning algorithm it is possible to use the BP with the modifications suggested by Elman and Jordan.

![Figure 2.9: Bengio NN combination of Jordan and Elman NN (picture based on Bengio 1993 [21])](image)

**2.4 Stochastic Methods**

Stochastic methods can be used in different parts and processes of NN learning algorithm. The reason for using it is to introduce randomness to the network. In the next paragraphs we will describe some possibilities for using the stochastic method. If the random factor is chosen properly in the used algorithm, it may improve the algorithm regarding accuracy and speed up the learning process. It
seems that the random factor has an important role on the plateau regions as well as in the local minimums. The main stochastic methods used in the NN are:

1. Echo State Networks (ESN),
2. Random Weights Initialization/Reinitialization,
3. Random Reordering of Patterns,
4. Random Selection Update,
5. Random Factor in Learning Constants.

### 2.4.1 Echo State Networks (ESN)

Echo state networks introduced by [28] belong to the supervised learning group of RCNN. The logic of ESN is to have a random, large fixed RCNN with an input signal, where neurons are in the so called “reservoir” of a network of nonlinear response signals, and combine a desired output signal by a trainable linear combination of all of these response signals (see Figure 2.10).

![ESN diagram](image)

Figure 2.10: The basic schema of an ESN, illustrated with a tuneable frequency generator task. Solid arrows indicate fixed, random connections; dotted arrows trainable connections (picture from Jaeger 2001 [28]).

The idea of ESNs is similar to the Liquid State Machines (LSM). The LSMs, ESNs and the more recently explored Backpropagation learning for RCNN (Schiller and Steil 2005) are subsumed under the name of Reservoir Computing. Schiller and Steil (2005) also showed that in traditional training methods for RNNs, where all weights (not only the output weights) are adapted, the dominant changes are
in the output weights. In cognitive neuroscience, a related mechanism has been investigated by Peter F. Dominey in the context of modelling sequence processing in mammalian brains, especially speech recognition in humans (e.g., Dominey 1995, Dominey, Hoen and Inui 2006).

The input signal $u(n)$ is a slowly varying frequency setting, the desired output $y(n)$ is a sinewave of a frequency indicated by the current input. The training input/output sequence $D = \{u(1), y(1)\} ... \{u(n_{max}), y(n_{max})\}$ is given. The task is to train an RCNN using training data such as that on slow test input signals. The output is again a sinewave of the input-determined frequency. The task using ESN is solved in 3 steps [28]:

1. **Creating random RCNN.** Create a random dynamical reservoir RCNN using any neuron model. The size of the reservoir is $N$ - depends on the task that is going to be solved. Attach input neurons to the reservoir by creating random full connections. Create output neurons. If the task requires output feedback, create randomly generated output-to-reservoir connections, otherwise do not create any connections to/from the output neurons in this step.

2. **Harvest reservoir states.** Drive the dynamical reservoir with the training data $D$ for times $n = 1, ..., n_{max}$. If the task does not require output feedback, the reservoir uses only input $u(n)$. The result is a sequence $x(n)$ of $N$ dimensional reservoir state. Each component signal $x(n)$ is a nonlinear transformation of driving input.

3. **Compute output weights.** Compute the output weights as the linear regression weights of the teacher outputs $y(n)$ on the reservoir $x(n)$. Use these weights to create reservoir-to-output connections (dotted connection on the Figure 2.10). The training is now complete and the ESN is ready for use.

### 2.4.2 Random Weights Initialization/Reinitialization

The random initialization of weights [29] with small values before the beginning of the learning NN should provide the distribution of values, according to which the weights are set. The range of values, which are valid for the weights reinitialized is mostly used from $(-4.0; 4.0)$ to $(-0.5; 0.5)$ [23]. The range depends on the task, the NN parameters which also can be involved, which is going to be solved if there is a way to involve external information into weight initialization. L. Bottou [30] in his works uses the range $(-\frac{a}{\sqrt{d_{in}}}, \frac{a}{\sqrt{d_{in}}})$. It is the simplest method of weight initialization. The different values of weights ensure, that during the learning process the weights will not have a tendency to assume identical values. The effectiveness of this approach depends on the initial weight distribution.
The experiment result [29] shows, that the random weight initialization approach is a suitable and convenient weight initialization method for a high MLP with identity activity function. For such an NN the most suitable initialization range is \(-0.014, 0.017\) also as shown in Figure 2.11. In the case that the hyperbolic tangent function is used as the activation function, the suitable range is \((-\frac{a}{\sqrt{d_{in}}}, \frac{a}{\sqrt{d_{in}}})\), where \(a\) is chosen so that the weight variance corresponds to one third of distance between the points of maximal curvature of the activation function. The shape of the initial weight distribution has very little effect on the optimal convergence time of the MLP. The main effect is a dislocation of the optimal value for the initial weight variant. The random reinitialization of weights can be used during the learning process, if the network gets stuck in local minimums and using standard methods is helpless [31].

![Figure 2.11: The average behavior of an MLP in convergence speed for changing the initial weight variance. Region A indicate the optimum initial weight variances that have been encountered, region B non-convergence (picture from Thim and Fiesler [29]).](image)

2.4.3 Random Reordering of Patterns

A widely used method that uses stochastic is randomly changing the sequence in which the patterns are presented during training. It is proposed as an alternative to sequential and batch processing for an on-line category of learning methods. It
provides enough diversity to the different networks in the ensemble. It is focused on the training set during the learning process [32]. The random training pattern reordering can be introduced before the training algorithm is applied or reordering the training set at the beginning of each iteration of the BP algorithm. The reordering of patterns during the training is a valid source for providing diversity to the networks. Upon detecting, that the categorization of a new input pattern is ambiguous, the input is postponed for a predefined time, after which it is re-examined and categorized for good. It shows that it improves the categorization performance over purely sequential processing, while yielding a shorter input response time, or latency, than batch processing. In order to examine the response time of processing schemes, the latency of a typical implementation is derived and compared to lower bounds.

2.4.4 Random Selection Update

Salvetti and Wilamovski [1] introduced in their study randomly selected weights that are updated. Instead of weights being updated in each step, their are updated one by one in random order. In this way, each time a pattern is presented not all weights are updated as in the classic BP algorithm but just the randomly selected weights. In order to keep track of the number of calculations involved in the process, one iteration is considered when the same number of weight updates occurred as if it was a standard BP iteration. The experiments were provided on the XOR problem with a regular multi-layered perceptron (MLP) with 2 neurons in the hidden layer. The results show that the classic BP was able to converge in 60% to 70% of cases, while with randomly selected weights they were able to converge in 85% of cases. Also the required number of iterations was reduced. The one iteration was defined as a cycle where all patterns are presented.

2.4.5 Random Factor in Learning Constants

An other Salvetti and Wilamovski [1] modification of the BP is using a random factor within the learning constant. Each time the derivative factor reaches a low threshold value, the learning constant $\gamma$ starts having a uniform random distribution. As the error gets more stable the wider range of the learning constant can be assumed. On the other hand, if the derivative has a large value, then the mean value of the learning constant is used. The average value of the learning constant remains the same, but the probabilistic distribution around this average increases and decreases. The learning constant $\gamma$ is computed from the following equation:

$$
\gamma = \gamma_0 + (\text{random}(2) - 1)\exp\left(-\frac{\partial E}{\partial w}\right)\text{random}\left(\frac{\partial E}{\partial w}\right).
$$

(2.31)
From the results of experiments we may observe a very significant improvement in the speed of learning.

2.5 Summary

Weather prediction is very important in today’s industries. The accuracy of predictions can decrease the amount of damage to the state economy as well as damage to individuals. To have precise information about weather, such as the amount of rain, which can cause huge floods like those which occurred in Slovakia in 2010, the power of the wind in a tornado’s area, the increase in temperatures in tropical oceans which can cause the powerful hurricanes, etc., can save many human lives. This chapter is an overview of the practical approach to prediction as well as from the point of view of the artificial neural networks and their ability to be used for it. In this chapter, I described the main modifications of the BPTT algorithm. The implementation of the classical BPTT algorithm is tricky and requires computer power. Simplifying it, can decrease the time required for implementation or the time required for producing a prediction from a large amount of data. Modifications of the BPTT algorithm arose from the specific requirements of the problems on, which it was used. It was also shown that the stochastic approach plays a role in the learning process. To combine these approaches can cause the NN to become more popular as predictors, and thus they will be widely used.
Chapter 3

Selected Methods and Proposed Algorithm Improvements

The implementation of a classical weight update in a backpropagation algorithm relies on all weights being updated in the order in which they are represented inside the NN. In classical MLP networks it is a simple layer-by-layer computation from inputs to the output layer. A less common, but still convenient, implementation is for a fully interconnected network, like in the common BPTT case. The necessity of updating weights from every neuron to every neuron can actually bring compact implementation - it is not required to care about any exceptions.

With sparse topologies, missing links often brings up some additional code into the implementation of handling them. With backpropagation the problem might become even more complicated as we have to care about the order of links. For instance, layered recurrent topology for the BPTT algorithm might become difficult to handle, as we have to unfold a relatively complex structure into several time steps and be able to evaluate it in both directions.

With Stochastic Weight Update the order of links is not important. In any topology, it just creates a pool of neurons with links, selects the number of them, and updates them in random order. This can simplify the link update part in the backpropagation implementations for complex topologies. With some future development, the Stochastic Weight Update might become a part of efficient backpropagation implementation for recurrent networks.

As the verification reference of experiments, the well known Backpropagation algorithm is chosen. The back-propagation algorithm can be implemented in several different ways. The Werbos implementation of the BP algorithm for the FFNN described in [5] and the BPTT algorithm for the RCNN described in [33] was used.

The modification of the BP algorithm was chosen for the Stochastic Weight Update [34]. It was introduced by Salvetti and Wilamowski in 1994 [1] in order to
speed up the NN convergence. This update method has also one other quality, its implementation is simple for arbitrary network topology. In the stochastic weight update scenario, a constant number of weights is randomly selected and updated. This is in contrast to the classical ordered update, where always all weights are updated. In the next chapter, I will describe the exact implementation, and present examples of results on toy-task data. The stochastic weight update is suitable for replacing the classical ordered update without any penalty on implementation complexity and with a good chance of not having a penalty on quality of convergence.

3.1 Prediction of Daily Temperature Profile Using Local Meteorological Data

We designed an application of artificial neural networks for local weather prediction. By implementations of appropriate network structure and proper selection of input/output signals, solid results were achieved. This application was the motivation for later experiments with recurrent neural networks.

3.1.1 Prediction System for Daily Temperature Profile

The system was implemented in the local heating company, where it was used to predict the daily temperature profile with a period of 15 minutes. Further, weekly and yearly profiles were predicted, as well as heat consumption profiles. The final prediction system consists of several chained neural networks and data processing modules. Training data for the neural networks were collected from meteorological stations around Košice city to capture the trend of the weather in this region. Additional training data were collected by web-robots from the internet from several weather forecast agencies. This weather prediction is taken as an ethalon to which the later results using the RCNN learned by the modified BPTT will be compared.

Local weather prediction with neural networks is based on the approximation of weather function by the black-box model from the weather data collected in a particular local region. This is the weather model for a single place on the map. Weather forecast agencies produce forecasts for bigger regions or even for continents or the whole globe [35]. The local prediction method is not practical for these agencies. However, for local companies, which are interested in the local weather course, this approximation-based weather prediction can be useful. It is especially flexible in producing the type of results not available from weather agencies. For instance it is possible to predict technology and weather connected
CHAPTER 3. PROPOSED ALGORITHM IMPROVEMENTS

Figure 3.1: Implementation of prediction system in TEKO infrastructure. The D2000 is the data management system which was available to connect to the sensors’ data and to the operators. The prediction system consists of seven neural network prediction modules and web-robots for collecting of weather forecast data from the internet.

signals, like heat consumption [36]. Also prediction within a 15-minute period is possible, and utilization of locally measured data is possible too. We can say that local weather prediction is an extension and refinement of the now generally available weather forecast.

In this case it was focused on the next day prediction of the air temperature within a 15-minute period. It was called the Daily Temperature Profile (DTP). It is the course of the temperature during a single day. Our predictor system will look into the weather data from the previous days and according to them it will produce the DTP prediction for tomorrow. In addition to the DTP prediction it provides weekly and yearly predictions. And in addition to temperature, the technology-related signals are predicted - the heat consumption in the city. Our data, which are going to be use in the prediction system, are company-owned weather and technological data, public data about the current weather condition in area, and publicly available weather forecast data for the next days.

Our system was implemented - for the local heating company Tepláreň Košice a.s. (TEKO) which provides heat for a significant part of the 230 000 citizens of Košice city. Most of the load is during the winter period, when it is used for the heating of buildings. The main variable for the heat demand is thus the air temperature. If we know the future temperature, all the heat signals (with some accuracy) might be derived from it. Using the predicted temperature, the company is able to heat up the whole system on time using just the required among of energy. Better accuracy of prediction brings economical and also ecological advantages, such as smaller costs of heating energy and the efficient usage of it, and in the end it saves Košice’s citizens money.

The prediction system had to be implemented into the already existing infras-
Table 3.1: List of neural networks in the prediction system

<table>
<thead>
<tr>
<th>Predictor</th>
<th>Outputs</th>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>NN1 DTP - 24hours / 15min. period</td>
<td>24 × 4 = 96</td>
<td>company data</td>
</tr>
<tr>
<td>NN2 DTP with web-robots inputs</td>
<td>24 × 4 = 96</td>
<td>company data + web robots</td>
</tr>
<tr>
<td>NN3 DTP using min/max data only</td>
<td>24 × 4 = 96</td>
<td>min/max data</td>
</tr>
<tr>
<td>NN4 Min/Max Temp. 365 days</td>
<td>365 × 2 = 730</td>
<td>past years data</td>
</tr>
<tr>
<td>NN5 Min/Max Temp. 7 days</td>
<td>7 × 2 = 14</td>
<td>past days data</td>
</tr>
<tr>
<td>NN6 Steam amount</td>
<td>24 × 4 = 96</td>
<td>company data and DTP</td>
</tr>
<tr>
<td>NN7 Steam power</td>
<td>24 × 4 = 96</td>
<td>company data and DTP</td>
</tr>
</tbody>
</table>

The system consists of seven neural networks (see list on Tab. 3.1). Some of them are used chained; for instance, - NN3 can be used to further process outputs from - NN4. Five networks are designed to predict the temperature for different time periods with different accuracy. We have a network for predicting the temperature for the whole year (365 days), for 7 days’ temperature, for the daily temperature profile with and without web-robot data, and for the daily temperature profile from minimal and maximal temperatures in the previous days. Two networks predict the steam amount and the amount of steam power used in heating processes per day within a 15-minute period.

All neural networks were standard feed-forward multilayer perceptron networks with one hidden layer. The number of input and output neurons varied according to the type of prediction and required time periods (see Tab. 3.2). The
three numbers under "topology" are the number of inputs, the number of hidden neurons, and the number of output neurons. "Training patterns 100 from 1458" mean, that 100 patterns from 1458 available were chosen for the training. "Iterations" is the number of learning cycles (through all the training data) during the training. The $\gamma$ parameter is the learning rate of the error backpropagation algorithm. The $\alpha$ is the momentum parameter for the momentum version of backpropagation, which was used.

The networks inputs are described in Tab.3.3. "Weather variables" are measured air temperatures, humidity, wind speed, solar irradiation and precipitation. "Technology variables" are the several temperatures, amounts and powers parameters from the heat production processes. "Date variables" describe the date and time of actual sample, the day is computed as the time-distance from the reference day in the middle of summer. The weather forecast data from the web-robots consist of min/max temperature forecasts from three agencies, and wind and humidity forecasts. We used linear and also nonlinear time-windows of data on the input, and the nonlinear sampling was, for instance, every hour from yesterday, every second hour from the day before, and 4 samples from 2days before.

The system was trained using various data, which could influence the temperature in the Košice region. To incorporate weather trends data into the process, data
### Table 3.3: Input data description (also see Tab.3.2 network topologies)

<table>
<thead>
<tr>
<th>No. of inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NN1</td>
<td>$36 \times 28 = 1008$ samples of 28 weather, technology and date variables</td>
</tr>
<tr>
<td>NN2</td>
<td>$144 \times 39 = 5616$ samples of 39 weather, technology and date variables</td>
</tr>
<tr>
<td>NN3</td>
<td>$3 \times 3 = 9$ min/max temperature and day-in-year ID from past 3 days</td>
</tr>
<tr>
<td>NN4</td>
<td>$2 \times 3 \times 52 = 312$ min/max temperatures for 52 weeks for 3 years</td>
</tr>
<tr>
<td>NN5</td>
<td>$3 \times 7 \times 3 = 63$ min/max temperature and day-ID for 3 weeks</td>
</tr>
<tr>
<td>NN6</td>
<td>$1008 + 96 = 1104$ NN1 inputs plus next day DTP data with 15min period</td>
</tr>
<tr>
<td>NN7</td>
<td>$1008 + 24 = 1032$ NN1 inputs plus DTP data with 1 hour period</td>
</tr>
</tbody>
</table>

from different meteorological stations around Košice were used, and in the network NN2 were also data collected from three weather forecast agencies through the internet using web-robots.

Web robots are implemented as system daemons, periodically (hourly) polling web-pages with forecasts. The downloaded pages are parsed by Java and Perl programs using syntactic analysis of html code. Extracted data are stored back into the system database, from where they are queried by predictor modules when needed.

#### 3.1.2 Experiments and Results

The whole data set for the NN1 network consists of measured values for the past 4 years. The training set was represented by 100 days chosen randomly from the whole data set. For testing the network performance all 4 years of data were used. The NN1 input is represented by a set of measured values of a couple of weather variables from 3 days from the history (yesterday, 1 day before yesterday, 2 days before yesterday). See tables Tab.3.1 and Tab.3.2 for details; Fig. 3.2 illustrates the performance of the network on one year of test data.

The results from the NN1 network are part of the input of the NN6 steam amount prediction network. The rest of the -NN6 inputs are the same as in the NN1. So, the accuracy of prediction of the NN6 is actually influenced by the accuracy of the NN1 DTP prediction. The -NN7 network for steam power prediction is related. It is similar to the previous NN6, the difference is only the network output - it is steam power. See Fig.3.3 for the performance plot of one year of data.

NN2 is the only network using web-robots data with weather forecasts. This should be an advantage over the -local-data-only network- NN1. Unfortunately, I haven’t had enough weather-forecast data, so this network was trained only for 2 months, and thus it is usable only for prediction during this summer period. Also
CHAPTER 3. PROPOSED ALGORITHM IMPROVEMENTS

Figure 3.3: Test of steam data prediction (networks NN6 and NN7) for one year of data from 1st of January to 31st of December, real versus predicted values. The error is displayed in the bottom parts of the frames.

its output is not as stable as that of networks trained using 4 years of data. See Fig.3.5 for an example of the different behaviors of NN2 and NN1. The training error for NN2 in Table 3.4 was 4.1% in 2 monthsof data and 500 learning cycles; for measuring the testing error I haven’t had enough data.

The NN3 has the simplest topology, its inputs are only the minimum and maximum temperatures of the 3 days before. The output is the same $24 \times 4$ as with the rest of the DTP predictors. This network is used to derive the daily course of temperature from the min/max temperature in the past days. This prediction is not as accurate as the other two DTP predictors, but it can be used for days in the
Figure 3.4: Test of NN3 for DTP prediction of one year of data using min/max temperature data only. The bottom curve is the error plot.

Figure 3.5: Example of the effect of using web-robots’ weather forecast data on the prediction. The dotted line is the basic prediction, based on the weather history for this day from past years and the weather in the past three days. This curve is smooth because it models the historical average for given conditions. However, it missed the sudden weather change at noon. The dashed web-robots’ prediction, although not as fitting in the morning, and not as smooth (probably because of the lack of training data), did model the second half of the day better by using the forecast information about possible weather changes.

The last two networks are for the middle-term and long-term predictions. The
Figure 3.6: Long-term prediction networks, NN5 for a 7-day DTP prediction and NN4 for a 365-day prediction. Both produce two values for every day of output – the minimal and maximal temperatures for this day. The plot for the 365-day predictor is split: on the left side are maximal temperatures for the whole year, and the right half are minimal temperatures for the whole year. The error is displayed in the bottom parts of the frames.

NN5 is used for a 7-day prediction. This network can be combined with the NN3 for DTP prediction from minimum and maximum data which themselves are produced by the NN5. The input for this network is the minimum and maximum temperatures from the past three weeks and the output is the predicted minimum and maximum temperatures for the next 7 days. See Fig. 3.6 for the performance.
illustration. For training this network, the collected temperatures for 12 years were used.

The last network NN4 is for long-term prediction. We are trying to predict the weather course for the whole year. The purpose is to support operators when they make preparations for the heating system for next year. This 365-day prediction can be used the same way as the 7-day prediction in combination with the NN3 network to produce DTP outputs with in a 15-minute period. This prediction also serves as a backup solution if there is some problem with data collection on-line or during the maintenance of the systems in the company – the operator will still have the same prediction available. The training data set consists of three years. Every year is represented by the minimum and maximum temperatures for every week. The current year’s prediction is derived from this year’s already recorded data, from the data from the past year and two years before. The missing data for the current year are substituted with the average values from previous years for particular weeks. See Fig. 3.6 for the NN4 performance illustration.

Table 3.4: Errors on training and on testing data

<table>
<thead>
<tr>
<th></th>
<th>Training</th>
<th>Testing</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>NN1</td>
<td>4.1%</td>
<td>11.4%</td>
<td>600</td>
</tr>
<tr>
<td>NN2</td>
<td>4.1%</td>
<td>-</td>
<td>500</td>
</tr>
<tr>
<td>NN3</td>
<td>7.7%</td>
<td>10.3%</td>
<td>400</td>
</tr>
<tr>
<td>NN4</td>
<td>10.2%</td>
<td>13.1%</td>
<td>300</td>
</tr>
<tr>
<td>NN5</td>
<td>12.8%</td>
<td>12.9%</td>
<td>300</td>
</tr>
<tr>
<td>NN6</td>
<td>17.7%</td>
<td>48.4%</td>
<td>300</td>
</tr>
<tr>
<td>NN7</td>
<td>10.5%</td>
<td>24.1%</td>
<td>300</td>
</tr>
</tbody>
</table>
CHAPTER 3. PROPOSED ALGORITHM IMPROVEMENTS

3.2 Stochastic Weight Update Method using Back Propagation Learning Algorithm on FFNN and RCNN

Simplification of implementation might bring some penalty on the quality of learning. However, Salvetti and Wilamowski shown on the XOR problem that the opposite is probably true, and the introduction of stochastic processes into the BP can actually improve the quality of convergence. The examination of the approach will be done again with the circle-in-square problem and possible variations will be discussed.

The proposed improvements will focus on implementation issues, the complexity of the algorithm will not be changed, and I will investigate the methods of providing performance similar to the original algorithm.

3.2.1 Stochastic Weight Update Algorithm

Salvetti and Wilamowski did incorporate stochastic processes into the backpropagation algorithm in order to improve convergence - to avoid getting stuck in local minima on the error surface [1]. Stochastic Weight Update was one off several ways how to do it. During learning, maybe it is not necessary to update all weights. Not all of them have the same influence on the network error. Some of them are more important or maybe have higher values, some of them are close to zero or maybe their influence is lower, or none. If we don’t update all weights, we can also save some computation time and thus maybe speed up the learning process.

ALGORITHM 3.5: Implementation of Backpropagation Algorithm for MLP

1. For $i = 0; i < \text{allRequiredIterations}$;
2. For $p = 0; p < \text{allPatternsCount}$;
3. propagateSignalThroughAllLayers();
4. countErrorSignalOfLayers();
5. adaptWeightsBasedOnErrorSignal();

Where $i$ is the iteration number (number of the learning cycle), and $p$ is the number of the presented pattern.
Stochastic Weight Update does not require heavy modification of the classical error backpropagation algorithm (see Algorithm 3.5). A slight difference is in the process of updating weights, after an error signal has been computed. A real random number is generated from the range \((0, 1)\). If its value is less than 0.5, then this weight will be updated, otherwise it will be left unmodified (Algorithm 3.7, for comparison the classical ordered weight update is Algorithm 3.6). There is a 50% chance that the weight is going to be selected and updated. The value from the range is chosen pseudo-randomly with approximately uniform distribution (like the random number generation in Java language [37]).

**ALGORITHM 3.6: Classical Ordered Weight Update for MLP**

```
adaptWeightsBasedOnErrorSignal()
1. For \(l = 0; l < \text{numOfLayers};\)
2. For \(w = 0; w < \text{allWeightsOfLayer};\)
3. adaptWeight();
```

Where \(l\) is the number of the layer, and \(w\) is the number of the weight to be updated.

**ALGORITHM 3.7: Stochastic Weight Update for MLP**

```
stochasticAdaptWeightsBasedOnErrorSignal()
1. For \(l = 0; l < \text{numOfLayers};\)
2. For \(w = 0; w < \text{allWeightsOfLayer};\)
3. if randomNum > 0.5;
4. adaptWeight();
```

Where \(l\) is the number of the layer, and \(w\) is the number of the weight to be updated. The randomNum is generated by a pseudo-random generator with uniform distribution, and the 0.5 represents a 50% chance that the current weight will be adapted. The 0.5 constant can be changed into any number from \((0, 1)\).

If it is required to increase the weight selection chance over 100% another loop can be added with the parameter, which controls how many times this random
CHAPTER 3. PROPOSED ALGORITHM IMPROVEMENTS

Selection probability during backpropagation of error signal and weights update

Figure 3.7: Visualisation of implemented algorithms using the stochastic factor. Each Stochastic approach uses the 50% ratio for the selected weight/\( \delta \) that is going to be updated. The stochasticity is implemented on different levels of the learning algorithm as they are described in the Figure. The arrows in Figure represents the flow of signal - propagated/backpropagated signal inside of the NN.
selection should be done. With raising this parameter, the probability that all weights are going to be selected and updated is increasing.

Salvetti and Wilamowski in their original paper didn’t prescribe the exact implementation, they just state: "Instead of being always updated in each step the weights are updated one by one in a random manner. In this way, each time a pattern is presented not all the weights are updated as usual but just the randomly selected ones.” Also noting: "In order to keep track of the number of calculations involved in the process, one iteration is considered when the same number of weight updates occur as if it was a standard backpropagation iteration.” Thus the actual implementation and also the methodology of the evaluation of results might differ.

The stochastic factor does not have to be placed just during the weight update, but also can participate in another part of the learning algorithm. There are also two places which can be used. First is the process during the weight update (Algorithm visualization Figure 3.7), which can participate during error signal calculating, which later influences the change over of delta weights that are going to be updated, plus its modification. These can be combined with the Stochastic Weight Update. The second option is to cancel the dependences of the neurons based on the layers. This means that the neuron will not take into consideration to which layer it belongs. There will be just a set of neurons (see Figure 3.7) which are going to be selected stochastically. The idea of the signal backpropagation is modified, not all neurons/weights will be updated, and the NN will keep a signal from the past. This Shuffle Delta Update gives no parameters how deep the history is going to be stored inside of nonupdated weights. This has a bigger meaning in NN, which contains a lot of weights, a lot of interconnected neurons, where not all weights need to be updated.

Figure 3.7 describes the BP algorithm with/without modification. There is no direct impact of stochasticity in the life process of the NN. Only through weights that have been adapted during the learning process. It is the same implementation of life process as the ordinal BP implementation. The idea is, that the NN with the stochastic factor in the learning process, creates separate paths from the input neurons through the weights to the output neurons with actual information contained inside of the weights on this path. The rest of the weights acts as a noise filter and keeps the NN more or less in a stable state, where the parameters of the NN do not have such an influence on the whole process of learning. The ordinal algorithm and modified algorithms that were proposed are:

1. Ordered Update: Classic BP algorithm (see Algorithm 3.5), where all weights are updated (Algorithm 3.7) in one iteration. The error signal is computed for all neurons without any preferences.
2. Stochastic Update: Modified BP algorithm, where not all weights are updated in the same iteration. The chance of a weight update is controlled by \( r \), where \( 0 \leq r \leq 100 \), which represents the probability of weight update in percentage. A weight is selected for update if the generated number is \( \text{rand} \leq r \). Otherwise the weight is not going to be updated. The NN keeps this way information from the past.

3. Stochastic \( \Delta \) Weight Update: This approach combines the Stochastic Update with the stochastic selection of \( \delta \) weights, which are going to be updated. After the back propagated error signal is computed, the \( \delta \) weights are going to be updated. The stochastic approach is the same as with the Stochastic Update. The \( \delta \) weight is updated if the \( \text{rand} \leq r \).

4. Stochastic \( \delta \) Update: Combines Stochastic Update with the stochastic selection of \( \delta \) Update. The stochastic factor is the same, as in the rest of the stochastic modifications. The \( \delta \) is selected to be updated with probability. This selective updating of \( \delta \) cases, that also have some old \( \delta \) from \( t - 1, ..., t - x \) (x - not known the past, that haven’t been updated yet) acts in the learning process and has an influence on the whole process.

5. Stochastic Complex Update: The stochastic factor of probability is the same as in the previous stochastic methods. It combines the Stochastic Update, the Stochastic \( \Delta \) Weight Update and the Stochastic \( \delta \) Update.

6. Shuffle \( \delta \) Update: This approach simplifies the implementation of the BP algorithm. A more obvious simplicity is in the fully connected- RCNN, where you do not have to care about layers and the order of the neurons. Simply one neuron is randomly selected from the ”pool of neurons”, the error signal is computed and all incoming weights are updated. The ”pool of neurons” represents all the neurons in all the layers, that contribute to the \( \delta \) calculation.


3.2.2 Stochastic Weight Update on FFNN

The Stochastic Weight Update was introduced by Salvetti and Wilamowski in 1994 [1] in order to improve the probability of convergence and the speed of convergence of the backpropagation algorithm. Besides the Stochastic Weight Update, they examined two other stochastic methods: random pattern selection and randomized learning rate. On the XOR problem they demonstrated significant improvement in the learning speed and probability of convergence for every one from these methods, especially for the randomized learning rate.

The backpropagation algorithm (Werbos, 1974; Rumelhart, McClelland, 1986) is one of the most used learning algorithms today. Plain vanilla implementation and momentum implementation [38] are the predominant implementations on feed-forward topologies. The multilayer perceptron (MLP) is most used feed-forward topology, it is a layered architecture with full connection between adjacent layers.

The Backpropagation Through Time (BPTT) [20] is the most used backpropagation variant for recurrent topologies. The BPTT was originally described, and is well suited, for recurrent topologies with full connectivity. Although the BPTT implementation is not more complex compared to vanilla backpropagation, the feed-forward time-delay networks are preferably used for time-related problems.

Sparse topologies became recently popular with the recurrent echo state networks (ESN), introduced by Jaeger [39]. However these are usually not based on the backpropagation algorithm. ESN networks are based on a so called reservoir of randomly interconnected neurons, so they have fully random sparse topology.

Pruning methods and cascade correlation [40] style algorithms for incremental building of topology can lead to pseudo-random sparse topologies too.

Although layered topologies are most frequent among backpropagation trained neural networks, many interesting structured or semi-random topologies are used too. Signal propagation through structured topologies might be non-trivial for more demanding backpropagation-group algorithms like the BPTT with time unrolling signal propagation. Stochastic Weight Update may simplify this "propagation" task.

3.2.3 Experiments of Stochastic Weight Update on FFNN

To test the Stochastic Weight Update [34], [41], the circle-in-square problem was chosen - a classification toy-task benchmark from the ART networks community (see Fig. 3.8). It was chosen to check that the Stochastic Weight Update will not reduce quality of backpropagation learning, and also to explore the influence of ratio of weights updated per pattern in the results. Besides the stochastic weights update, the stochastic factor was also involved in the delta weight selection and
delta update. One special case is the shuffle delta update, where the layers are not taken into consideration and the update of neurons is selected from a pool of neurons. The selected neuron is randomly chosen. The random number is from interval $\langle 0, N \rangle$, where $N$ is aggregation of neurons from the hidden layer and the output layer.

![Figure 3.8](image-url)

Figure 3.8: Visualization of the circle-in-square problem. The task is to classify vectors of $x$ and $y$ coordinates into two classes - those which belong in circle and those which are outside.

The classifier, the neural network, has to classify the incoming data, pairs of $x$ and $y$ coordinates, into two classes: circle class, square class. The coordinates of the points for circle class and, square class are $x, y \in \langle 0, 1 \rangle$. Available data are divided into the training set and the testing set. In the training set I had 1001 samples, 492 of which represented the square class and 509 of which represented the circle class. The testing data set consisted of 10002 samples, the square class was represented by 5004 samples, and the circle class was represented by 4998 samples. The visualization of the data set is in Fig. 3.8.

The Stochastic Weight Update will be compared with the classical ordered update on a multilayer perceptron with the backpropagation algorithm. In addition further stochastic selection on the level of the neurons during the computation of delta weights will be involved. The same number of neurons will be used, the same sigmoid function with the steepness $\lambda$ and the same learning rate $\gamma$. The topology consists of one hidden layer, two input neurons, which represent $x$ and $y$ coordinates of the points, and one output neuron, which classifies the required class. Networks are trained with an online learning method - after each sample (point) the error signal is computed and the weights are updated.

In this case for all networks, one iteration during the learning process means, that after each training sample, the error signal is calculated and then the weights are updated. In the stochastic implementation not all weights are updated in a single iteration (only the selected ones) and, some weights keep their values from the previous state. All the experiments were run ten times with the same settings. For the result of the experiment, the best result is chosen and this is used to create the output figure.
In the first experiment (Fig. 3.9, Table 3.8, Table 3.9) the Stochastic Weight Update is compared with the regular update on a well trained network and with the next 4 mentioned stochastic factors implementation. All results are comparable in quality. All networks have the same settings.

The second experiment is presented to show the effect of the ratio of weights updated, the shuffle neuron selection or the influence of the stochastic factor in whole network per pattern in the Stochastic Weight Update. The parameters of learning are different from the first experiment. The setup is positioned into a situation where the classical weight update was not able to achieve comparable results with the Stochastic Weight Update using the same number of hidden neurons. So the presented classical weight update network has more hidden neurons. Other network parameters are the same for both networks. The number of iterations is the same 2600 compared to previous experiment.

The chosen levels of the ratio are 25%, 50%, 75% and 100%. Note that ratio 100% is actually the same behavior as if using the classical ordered weight update. Classification results are displayed in Table 3.10. The best results were actually
Figure 3.11: Visualization of the result of well trained networks with the Stochastic Weight Update Delta Update and Stochastic Weight Update combined with the Shuffle Delta Update with the backpropagation algorithm with the same settings. Results are comparable.

achieved with the 50% ratio (see also Fig. 3.12 for the visualization of the obtained results). The course of the training is illustrated in Figure 3.13. The overall course of the training seems similar for all ratios except 100% when the algorithm got stacked. Note that the actual amount of computations per iteration is different for every ratio.

3.2.4 Results of Stochastic Weight Update Algorithm on FFNN

To provide the sufficient experiment results, each experiment was executed 10 times. The best results of the experiments was selected and used as the final representative results.

In this case, the network with the Stochastic Weight Update was able to learn and the accuracy is comparable to the classical ordered update in the backpropagation algorithm. The rest of the networks with the stochastic factor prove, that they are also able to learn and show comparable results as the network with the Ordered and Stochastic Weight Update.

Results of $\gamma$ Resistance

The experience shows, that the classical ordered weight update is more sensitive to the changes of the learning rate $\gamma$. If the learning rate is greater, the ordered update requires more neurons in the hidden layer (see the second experiment and Table 3.10).
Table 3.8: Classical Ordered vs. Stochastic Weight Update

<table>
<thead>
<tr>
<th>Weight Update</th>
<th>Ordered</th>
<th>Stochastic (50%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma$</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Topology</td>
<td>2-5-1</td>
<td>2-5-1</td>
</tr>
<tr>
<td>Iterations</td>
<td>2600</td>
<td>2600</td>
</tr>
</tbody>
</table>

Accuracy of classification

<table>
<thead>
<tr>
<th></th>
<th>Circle Points</th>
<th>Square Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical</td>
<td>98.14%</td>
<td>99.44%</td>
</tr>
<tr>
<td>Stochastic</td>
<td>97.14%</td>
<td>99.26%</td>
</tr>
</tbody>
</table>

Where $\gamma$ = learning rate, $\lambda$ = steepness of the sigmoid function, Topology = topology of the network with one hidden layer, Iterations = how many times the training set was presented during the process of learning. The Stochastic Weight Update was run with the constant 0.5 in Algorithm 3.7 (it is 50% selection ratio). Presented accuracy values are the best from ten runs.

Figure 3.12: Visualization of the result of networks with different selection ratios in the Stochastic Weight Update using settings from Table 3.10. The 100% selection ratio is actually equal to the classical ordered weight update. Using this 100% ratio it was unable to obtain good results.
Table 3.9: Stochastic factor in BP on different levels

<table>
<thead>
<tr>
<th>Weight Update</th>
<th>Stochastic</th>
<th>delta weight</th>
<th>delta</th>
<th>delta n.2</th>
<th>shuffle delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma$</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Topology</td>
<td>2-5-1</td>
<td>2-5-1</td>
<td>2-5-1</td>
<td>2-5-1</td>
<td>2-5-1</td>
</tr>
<tr>
<td>Iterations</td>
<td>2600</td>
<td>2600</td>
<td>2600</td>
<td>2600</td>
<td>2600</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accuracy of classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle points</td>
</tr>
<tr>
<td>square points</td>
</tr>
</tbody>
</table>

Where $\gamma =$ learning rate, $\lambda =$ steepness of the sigmoid function, Topology = topology of the network with one hidden layer, Iterations = how many times the training set was presented during the process of learning. The Stochastic Weight Update was run with the constant 0.5 in Algorithm 3.7 (it is 50% selection ratio).

Presented accuracy values are the best from ten runs. Stochastic = stochastic weight updates, delta weight = stochastic weight update and stochastic $\delta$weight update, delta = stochastic weight update, stochastic $\delta$weight update and stochastic $\delta$ update, delta n.2 = stochastic weight update and stochastic $\delta$ update, shuffle delta = stochastic weight update and shuffle $\delta$ update.
### Table 3.10: Influence of Probability Selection of Weights to be Updated

<table>
<thead>
<tr>
<th>Weight Update</th>
<th>Stochastic</th>
<th>Ordered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Ratio</td>
<td>25%</td>
<td>50%</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>$\lambda$</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>Topology</td>
<td>2-5-1</td>
<td></td>
</tr>
<tr>
<td>Iterations</td>
<td>2600</td>
<td></td>
</tr>
<tr>
<td>Accuracy of classification</td>
<td></td>
<td></td>
</tr>
<tr>
<td>circle points</td>
<td>96.1%</td>
<td>96.6%</td>
</tr>
<tr>
<td>square points</td>
<td>97.9%</td>
<td>98.9%</td>
</tr>
</tbody>
</table>

Variable names are the same as in Table 3.8, the accuracy values are again the best obtained from ten runs of learning. The 100% selection ratio in the stochastic update (by Algorithm 3.7) is actually equal to the ordered update, it means 100% of the links were updated. See also the visualized results in Fig.3.12.
Figure 3.13: The course of the mean squared error of the Stochastic Weight Update backpropagation with different selection ratios. The overall course of training seems similar for all ratios except 100%, when the algorithm got stacked. Courses are smoothed for better readability. This graph corresponds to the results in Table 3.10 and in Figure 3.12.
3.2.5 Experiments of Stochastic Weight Update on RCNN

The time-shifted and skewed signal (a toy-task benchmark; see Figure 3.14) was used to compare different types of stochastic implementation (see Figure 3.7) with ordered implementation of the BPTT algorithm on an RCNN, as a test of the correct algorithm implementation and as a test of the Stochastic Weight Update which does not reduce the quality of backpropagation learning. This toy-task benchmark is suitable for exploring the influence of ratio of weights updated per pattern in the results. In the experiments the same number of neurons were used, the same sigmoid function with the steepness $\lambda$ and the same learning rate $\gamma$. The results are compared to the lowest least mean square error (LMS). Experiments were provided also using weather data (data used for the TEKO TDP prediction) to do a prediction of TDP.

![Figure 3.14: Toy-task benchmark time-shifted and skewed signal to prove the correct implementation of RCNN. The input represents the input signal to the NN. The evi output represents expected signal on the output of the NN.](image)

To provide the sufficient experiment results, each experiment was executed 10 times. The best results of the experiments was selected and used as the final representative results.

3.2.6 Prediction of Temperature Daily Profile

This experiment is focused on the Temperature Daily Profile (TDP) prediction [42]. It seems quite easy due to the general model trend of temperature which is more or less the sine curve with a slight deformations. The dataset used for the TDP
CHAPTER 3. PROPOSED ALGORITHM IMPROVEMENTS

prediction consists of the measured temperatures from the year 2000 till the end of February, 2011. The data are measured each half hour, which provides a wide range of options for creating suitable training and testing data for the RCNN. This wide range also allows the RCNN to create an internal model of the TDP. The data were collected by the web page Weather Underground (WUD) [43]. The dataset contains information about the Date of collection (CET), the Time (CET) of collection, Current Temperature, Dew Point, Humidity, Sea Level Pressure, Visibility, Wind Direction, Wind Speed, Gust Speed, Precipitation, Events like Fog; Snow; Rain...and Conditions like ScatteredClouds; PartlyCloudy; Clear... . In this experiment only the temperature and timewere used more or less in a hidden format - a sequence of data collection. The dataset is split into two groups. The training dataset, which is provided to the network in the learning phase, and the testing dataset, which is an unknown part of the dataset to the NN and is used to test the ability of the NN to predict the temperature and the ability to generalize the temperature model hidden in the temperature daily profile dataset.

The idea of TDP prediction is to predict the 24-hour TDP in a head for the next day. Only the prediction for the next couple of hours was used to test the RCNN modification. The input into the RCNN is one hour. The output is the TDP for the next 24 hours. The missing window set/history of the data between is stored in the recurrent connections of the network based on the unfolding fundamentals of the BPTT algorithm. In such a case, it is not required to use an input window set.

To prove a workable prediction, a simple prediction test was proposed, which consists of the predicting TDP for next 3 hours. The training set consists of the 3 winter months of 2006 - October, November, and December, with 1 hour sampling. The test set was the month of January, 2007. The input parameter is one hour. The history of the unfolding RCNN is set to 2 with 13 hidden neurons. For closer settings see Table 3.11. Fig. 3.16 represents the LMS of the RCNNs and samples of the prediction are presented in Fig 3.15.

The prediction for a longer period of time requires more time to learn the RCNN, to analyze the behavior of the RCNN, to think about in which form the data should be used. It is necessary to provide deeper analyzes of the behavior of the RCNN with more complex data than the toy-task benchmark (Fig. 3.14). It seems, based on the current experiment results, that the RCNN is trying to generalize the temperature for the rest of the prediction time, and is focused more on the prediction of the next data set. Also the problem with the RCNN is that the RCNN gets stacked in many local minima, in which cases that RCNN is not able learn the proper way and has to be reset. The dynamic system of the network becomes more difficult and the stability-plasticity dilemma is more obvious [44].
Figure 3.15: Sample of result predictions for one day of using the Ordered update and the Shuffle δUpdate. The rest of graphs are on the enclosed CD.
Figure 3.16: LMS error of the Prediction of the Temperature Daily Profile experiment.
Table 3.11: Temperature Daily Profile prediction

<table>
<thead>
<tr>
<th>Method</th>
<th>Ordered Update</th>
<th>Stochastic Update</th>
<th>Stochastic ∆Weight Update</th>
<th>Stochastic Complex Update</th>
<th>Stochastic δUpdate</th>
<th>Shuffle δUpdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Ratio</td>
<td>100%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>γ</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>λ</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>α momentum</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Unfolding history T</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Topology</td>
<td>1-13-1</td>
<td>1-13-1</td>
<td>1-13-1</td>
<td>1-13-1</td>
<td>1-13-1</td>
<td>1-13-1</td>
</tr>
<tr>
<td>Adaptation in one cycle</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Iterations</td>
<td>2500</td>
<td>2500</td>
<td>2500</td>
<td>2500</td>
<td>2500</td>
<td>2500</td>
</tr>
<tr>
<td>LMS error</td>
<td>6.08</td>
<td>4.69</td>
<td>7.03</td>
<td>3.41</td>
<td>9.21</td>
<td>5.34</td>
</tr>
</tbody>
</table>

This table represents the settings of the RCNNs for the TPD prediction. All RCNNs have the same settings. It is a comparison of the modified RCNN vs. the unmodified RCNN.
3.2.7 Stochastic Updates vs. Ordered Update

This experiment consists of 5 RCNNs using different modifications of the stochastic update (see Figure 3.7). The topology of all the RCNNs is based on the best results of the ordered update. Figure 3.17 represents the results of the first experiment; the error is presented in Figure 3.20 and the detailed settings of the RCNNs are in Table 3.12. The experiment was focused on proving the learning ability of the stochastic modifications and achieving better results using one of the stochastic modifications compared to the origin update of the BPTT implementation on the RCNN. The rest of the graphs are on the enclosed CD.
Figure 3.18: LMS error of the experiment based on the best of the ordered update of the BPTT.
Table 3.12: To achieve better results than the Ordered Update algorithm of the BPTT

<table>
<thead>
<tr>
<th>Method</th>
<th>Ordered Update</th>
<th>Stochastic Update</th>
<th>Stochastic Update</th>
<th>Stochastic Update</th>
<th>Stochastic Update</th>
<th>Stochastic Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Ratio</td>
<td>100%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>γ</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>λ</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
</tr>
<tr>
<td>α momentum</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Unfolding history T</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Adaptation in one cycle</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Iterations</td>
<td>1500</td>
<td>1500</td>
<td>1500</td>
<td>1500</td>
<td>1500</td>
<td>1500</td>
</tr>
<tr>
<td>LMS error</td>
<td>0.00572</td>
<td>0.01063</td>
<td>0.00449</td>
<td>0.0106</td>
<td>0.197</td>
<td>0.00071</td>
</tr>
</tbody>
</table>

This table represents the settings of the RCNNs for the experiment of the Stochastic Update modifications vs. the Ordered Update - achieving better results than the ordered update of the BPTT. All RCNNs have the same settings. The Shuffle δ Update modification was able to achieve a higher accuracy.
3.2.8 Stochastic Updates vs. Ordered Update Including the $\alpha$ Momentum

The experiment that involves the $\alpha$ momentum [45] for the RCNN is a little bit different from the previous experiment. It uses the $\alpha$ momentum to get better results and does not rely just on the algorithm ability to deal with postponed signal. This experiment was focused on the influence of the $\alpha$ momentum on the stochastic algorithm modifications, to use the $\alpha$ as a final touch to improve learning of the RCNN after the desirable topology was found. Figure 3.19 represents the results of the experiment, error is presented on Figure 3.20 and detailed settings of RCNNs are presented in Table 3.13.

![Figure 3.19: Output of the RCNNs for the BPTT using the $\alpha$ momentum. The best results were achieved by the Stochastic Update modification (see Table 3.12).](image)
Figure 3.20: LMS error for the BPTT using the $\alpha$ momentum. The best results were achieved by the Stochastic Update modification (see Table 3.12).
Table 3.13: Using the $\alpha$ momentum for the SRCNN

<table>
<thead>
<tr>
<th>Method</th>
<th>Ordered Update</th>
<th>Stochastic Update</th>
<th>Stochastic $\Delta$Weight Update</th>
<th>Stochastic Complex Update</th>
<th>Stochastic $\delta$Update</th>
<th>Shuffle $\delta$Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Ratio</td>
<td>100%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
</tr>
<tr>
<td>$\alpha$ momentum</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
</tr>
<tr>
<td>Unfolding History T</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Adaptation in one cycle</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Iterations</td>
<td>2500</td>
<td>2500</td>
<td>2500</td>
<td>2500</td>
<td>2500</td>
<td>2500</td>
</tr>
<tr>
<td>LMS error</td>
<td>0.00107</td>
<td>0.00098</td>
<td>0.00145</td>
<td>0.00166</td>
<td>0.00133</td>
<td>0.00141</td>
</tr>
</tbody>
</table>

Experiment results table. The experiment focused on the usage of the $\alpha$ momentum to achieve higher accuracy. The best results were achieved by the Stochastic Update modification.
3.2.9 Influence of Learning Parameter $\gamma$ on Stochastic Update

This experiment is focused on the resistance of the learning parameter $\gamma$ for the stochastic modifications of the RCNN ([46]). The amount of the influence of $\gamma$ on the modifications. The results of the Stochastic Update modifications are compared to the origin Ordered Update implementation of the BPTT algorithm. The same number of neurons was used and, the same sigmoid function with the steepness $\lambda$. The learning parameter $\gamma$ is changed by the experiment from 0.03, 0.1, 0.3, 0.5, 0.7 to 0.9, together with six experiments with different values of the learning parameter $\gamma$. Figure 3.21 represents the graphical results of the experiment and the detailed settings the of RCNNs are in Table 3.14. Figure 3.22 represents the Least Mean Square error (LMS) which is used as a comparison of the $\gamma$ experiments. A lower LMS provides a better result. The rest of the graphs from the experiment are on the enclosed CD.
Figure 3.21: The influence of the learning parameter $\gamma$ on the learning process using Ordered Update. The influence of the learning parameter $\gamma$ on the learning process using Stochastic Update. The influence of the learning parameter $\gamma$ on the learning process using Shuffle $\delta$ Update.
Figure 3.22: LMS error of the influence of the learning parameter $\gamma$ on the learning process using Ordered Update. LMS error of the influence of the learning parameter $\gamma$ on the learning process using Stochastic Update. LMS error of the influence of the learning parameter $\gamma$ on the learning process using Shuffle $\delta$ Update.
### Table 3.14: Influence of $\gamma$ on Stochastic Update modifications

<table>
<thead>
<tr>
<th>Method</th>
<th>Ordered Update</th>
<th>Stochastic Update</th>
<th>$\Delta$Weight Update</th>
<th>Stochastic Complex Update</th>
<th>Stochastic $\delta$Update</th>
<th>Shuffle $\delta$Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Ratio</td>
<td>100%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>$\gamma$ 1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>$\gamma$ 2</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>$\gamma$ 3</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>$\gamma$ 4</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>$\gamma$ 5</td>
<td>0.9</td>
<td>0.9</td>
<td>0.9</td>
<td>0.9</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
</tr>
<tr>
<td>$\alpha$ momentum</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Unfolding history T</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Adaptaptation in one cycle</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Iterations</td>
<td>1500</td>
<td>1500</td>
<td>1500</td>
<td>1500</td>
<td>1500</td>
<td>1500</td>
</tr>
<tr>
<td>LMS error 1</td>
<td>0.0271</td>
<td>0.0081</td>
<td>0.0082</td>
<td>0.012</td>
<td>0.017</td>
<td>0.0020</td>
</tr>
<tr>
<td>LMS error 2</td>
<td>0.0216</td>
<td>0.013</td>
<td>0.0310</td>
<td>0.0073</td>
<td>0.0035</td>
<td>0.0033</td>
</tr>
<tr>
<td>LMS error 3</td>
<td>0.0071</td>
<td>0.0247</td>
<td>0.0052</td>
<td>0.0086</td>
<td>0.017</td>
<td>0.0027</td>
</tr>
<tr>
<td>LMS error 4</td>
<td>0.0542</td>
<td>0.0504</td>
<td>0.0027</td>
<td>0.0212</td>
<td>0.0016</td>
<td>0.0030</td>
</tr>
<tr>
<td>LMS error 5</td>
<td>0.0341</td>
<td>0.0063</td>
<td>0.0044</td>
<td>0.0017</td>
<td>0.0022</td>
<td>0.0066</td>
</tr>
</tbody>
</table>

The influence of the learning parameter $\gamma$ on the learning process. The stochastic update modifications are less sensitive to the $\gamma$, than the ordered update.
3.2.10 Influence of the Number of Recurrent Neurons in Time Delay RCNN

The Time Delay RCNN (TDRCNN) is an RCNN with one of six types of implemented algorithms provided in this master’s thesis (see Figure 3.7) with a window consisting of time series data ([47]) as the input. The experiment was set up in based on the results of a TDP prediction experiment using the RCNN to discover the influence of a different number of recurrent neurons in the TDRCNN. In the TDP prediction experiment the RCNN was getting stacked in local minima and due to a more complicated dynamic system of RCNN than the FFNN, it was difficult to find out a suitable topology and network settings. By the recurrence in the network the network becomes more simple, which provides more room for manipulation of network parameters and topology. Without recurrent neurons the TDRCNN becomes regular TDNN and adding recurrent neurons, the TDRNN becomes more complicated. A regular RCNN is more likely get stacked in local minima with a full connection than with simplified topology. The experiment was realized with the same topology as previous experiments. The RCNN consists of 15 hidden neurons, 1 output neuron and 4 input neurons. Toy-data and a time-shifted and skewed signal were used. The 4 input neurons represent the time window on the signal. The results in table 3.15 were chosen as an experiment with the lowest LMS error from 10 running experiments of one type of network modification. The modifications of TDRCNN lie in the reduction of recurrent neuron’s number from the full connection through 10, 5, 2, 1 and no recurrent neurons, which represents a regular feed forward TDNN. The recurrent neurons were chosen randomly from a pool of all neurons. Once the neuron was chosen, it was not taken into consideration for the next random neuron choice. The selected neurons were taken into consideration in the learning process - the adaptation of weights. The results are in table 3.15. Provided figures illustrate two kinds of comparisons: the comparison of selected neurons in one type of algorithm Figure 3.23 with the logarithmic LMS error in Figure 3.24, compared to the selected number of neurons through all types of algorithm modifications (Figure 3.25) with the logarithmic LMS error in Figure 3.26. The rest of the figures can be found on the enclosed CD.
CHAPTER 3. PROPOSED ALGORITHM IMPROVEMENTS

Figure 3.23: The influence of recurrent neurons on the learning process using Ordered Update. The influence of recurrent neurons on the learning process using Stochastic Update. The influence of recurrent neurons on the learning process using Shuffle δ Update.
Figure 3.24: LMS error of the influence of recurrent neurons on the learning process using Ordered Update. LMS error of the influence of recurrent neurons on the learning process using Stochastic Update. LMS error of the influence of recurrent neurons on the learning process using Shuffle δ Update.
Figure 3.25: The influence of recurrent neurons on the learning process using 0 recurrent neurons. The influence of recurrent neurons on the learning process using 1 recurrent neurons. The influence of recurrent neurons on the learning process using all recurrent neurons.
Figure 3.26: LMS error of the influence of recurrent neurons on the learning process using 0 recurrent neurons. LMS error of the influence of recurrent neurons on the learning process using 1 recurrent neurons. LMS error of the influence of recurrent neurons on the learning process using all recurrent neurons.
Table 3.15: Influence of different numbers of recurrent neurons in Time delay RCNN (TDRCNN)

<table>
<thead>
<tr>
<th>Method</th>
<th>Ordered Update</th>
<th>Stochastic Update</th>
<th>Stochastic $\Delta$Weight Update</th>
<th>Stochastic Complex Update</th>
<th>Stochastic $\delta$Update</th>
<th>Shuffle $\delta$Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Ratio</td>
<td>100%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
<td>0.65</td>
</tr>
<tr>
<td>$\alpha$ momentum</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Unfolding history $T$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Adaptation in one cycle</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Iterations</td>
<td>1500</td>
<td>1500</td>
<td>1500</td>
<td>1500</td>
<td>1500</td>
<td>1500</td>
</tr>
<tr>
<td>LMS 0 RCN*</td>
<td>$19.51 \times 10^{-4}$</td>
<td>$33.0 \times 10^{-4}$</td>
<td>$41.29 \times 10^{-4}$</td>
<td>$34.22 \times 10^{-4}$</td>
<td>$59.52 \times 10^{-4}$</td>
<td>$293.9 \times 10^{-4}$</td>
</tr>
<tr>
<td>LMS 1 RCN*</td>
<td>$27.45 \times 10^{-4}$</td>
<td>$25.43 \times 10^{-4}$</td>
<td>$26.37 \times 10^{-4}$</td>
<td>$65.21 \times 10^{-4}$</td>
<td>$41.47 \times 10^{-4}$</td>
<td>$57.12 \times 10^{-4}$</td>
</tr>
<tr>
<td>LMS 2 RCN*</td>
<td>$21.6 \times 10^{-4}$</td>
<td>$26.38 \times 10^{-4}$</td>
<td>$64.85 \times 10^{-4}$</td>
<td>$35.85 \times 10^{-4}$</td>
<td>$48.53 \times 10^{-4}$</td>
<td>$115.9 \times 10^{-4}$</td>
</tr>
<tr>
<td>LMS 5 RCN*</td>
<td>$13.4 \times 10^{-4}$</td>
<td>$37.68 \times 10^{-4}$</td>
<td>$66.95 \times 10^{-4}$</td>
<td>$14.72 \times 10^{-4}$</td>
<td>$25.69 \times 10^{-4}$</td>
<td>$1321 \times 10^{-4}$</td>
</tr>
<tr>
<td>LMS 10 RCN*</td>
<td>$13.39 \times 10^{-4}$</td>
<td>$28.77 \times 10^{-4}$</td>
<td>$116.5 \times 10^{-4}$</td>
<td>$575.1 \times 10^{-4}$</td>
<td>$48.49 \times 10^{-4}$</td>
<td>$619.4 \times 10^{-4}$</td>
</tr>
<tr>
<td>LMS all RCN*</td>
<td>$73.29 \times 10^{-4}$</td>
<td>$262.9 \times 10^{-4}$</td>
<td>$539.7 \times 10^{-4}$</td>
<td>$337.1 \times 10^{-4}$</td>
<td>$443.7 \times 10^{-4}$</td>
<td>$2527 \times 10^{-4}$</td>
</tr>
</tbody>
</table>

Influence of the chosen different numbers of recurrent neurons in Time delay RCNN. RCN* - number of chosen recurrent neurons during the learning process.
3.2.11 Results of Stochastic Weight Update on RCNN

The results with the time-shifted and skewed signal confirm the original results of Salvetti and Wilamowski on the XOR problem. Experiments provided on the RCNN with time-shifted - the Stochastic Weight Update performs well and in some cases can be superior to the classical ordered update in the backpropagation through time algorithm.

The Stochastic Update modifications in different variants is less sensitive to the different learning parameter $\gamma$ compare to the classical ordered update. Still there is some influence of the $\gamma$ which depends on the implementation level of stochasticity. This can be used as an advantage during the tuning of the RCNN and we can be more focused on the topology itself and the data structure used for the RCNN. To find out the suitable $\gamma$ value can be postponed till we have a more or less known working topology. In some cases, this less dependence of RCNN on the $\gamma$ can theoretically improve the accuracy of the results produced by the RCNN.

Provided weather experiments on RCNN show, that the RCNN can easily stuck in the local minima and it is difficult to find out the proper topology with the corresponding parameter settings. This leads us to provide experiments where the RCNN contained the window as an input and the recurrent neurons were added step by step from 0 - regular feed forward TDNN through 1,2,5,10 and full connection of recurrent neurons. The results, represented in table 3.15, show that using recurrent neurons in small amount of all neurons, 13% - 33%, improves the accuracy of the network compared to the regular feed forward TDNN and full TDRCNN.
Chapter 4

Scientific Contribution

4.1 Summary of the Contribution

The scientific contribution can be summarized as follows:

- Learning process of the Stochastic Weight Update is comparable to original BPTT (see section 3.2.3 page 49 - 3.2.4 page 52), Table 3.8.
- Performance of algorithm is less dependent on the network topology (see section 3.2.3 page 49, 3.2.4 page 52), Table 3.9 and Table 3.10.
- It has higher $\gamma$ resistance (section 3.2.9 page 68 and 3.2.4 page 52)
- Possible simplifying implementation on some RCNN topologies (section 3.2.10 page 72)

The summary of the main result of this work are:

Daily Temperature Profile Prediction

Achieved prediction accuracies are shown in Table 3.4 (see section 3.1 page 35). The training errors are measured after 300 - 600 training cycles. Testing errors, although more important than the training errors, are sometimes measured on a bigger testing set, sometimes on a smaller one, it depend on how much data we had available for the particular type of prediction. For the NN2 we didn’t have enough data to build any meaningful testing data set. Considering the type of prediction of every network, the NN1 predictor is the most accurate one, and we used it as the primary source of prediction for the heating company. Combinations of other predictors are used when the NN1 cannot be used. In the future, when there will be more weather forecast data collected, we believe the NN2 might outperform the NN1 predictor.
During the experiments we found the temperature to be the most influencing factor of all the input data concerning prediction accuracy. In the time-windows, the most influencing part were the last three-four hours of the past temperature course. The technology variables were in general less influencing compared to the weather variables, however, if we skipped them the accuracy decreased a little.

**Feed Forward Neural Networks Summary**

The results with the circle-square task (see section 3.2.3 page 49) confirm the original results of Salvetti and Wilamowski on the XOR problem - the Stochastic Weight Update performs well and in some cases can be superior to the classical ordered update in the backpropagation algorithm.

In experience, the Stochastic Weight Update and their modifications (see Figure 3.7 page 46) based on stochastic factor in the BP algorithm are also less sensitive to the settings of the learning rate, compared to the classical ordered update (see section 3.2.4 page 52). It can be used in cases where it is not easy to find out the working configuration for the network.

The main goal is to bring forward the implementation possibilities which the Stochastic Weight Update might open. The complexity of implementation of this weight update is less dependent on the network topology (see experiments 3.2.3 page 49) than the classical ordered weight update is, while overall complexity of implementation is not much higher than the classical one. This can be utilized in backpropagation implementations with non-standard, sparse, or random network topologies.

**Recurrent Neural Networks Summary**

The results with the time-shifted and skewed signal (section 3.2.6 page 57 - 3.2.11 page 78) confirm the original results of Salvetti and Wilamowski on the XOR problem. The experiments provided on the FFNN using the circle-square separation task and also on the RCNN with a time-shifted and skewed signal - the Stochastic Weight Update performs well and in some cases can be superior to the classical ordered update in the backpropagation through time algorithm.

The complexity of implementation of this weight update is less dependent on the network topology than the classical ordered weight update is, while overall complexity of implementation is not much higher than the classical one. This can be utilized in BPTT implementations with non-standard, sparse, or random network topologies.

The conclusion of the experiment is, that the Stochastic Update modifications in different variants is less sensitive to the different learning parameter $\gamma$ (section 3.2.9 page 68), $\alpha$ momentum (section 3.2.8 page 65) compared to the classi-
cal ordered update (section 3.2.7 page 62). Still there is some influence of the \( \gamma \) which depends on the implementation level of stochasticity. This can be used as an advantage during the tuning of the RCNN and we can be more focused on the topology itself and the data structure used for the RCNN. Finding out the suitable \( \gamma \) value can be postponed till we have a more or less known working topology. In some cases, this less dependence of the RCNN on the \( \gamma \) can theoretically improve accuracy of the results produced by the RCNN.

The provided weather experiments on the RCNN show (section 3.2.6 page 57), that the RCNN can get easily stuck in the local minimum and it is difficult to find out the proper topology within the corresponding parameter settings. That was the reason for providing experiments where the TDRCNN did contain the window as an input and the recurrent neurons were added step by step from 0 - regular feed forward TDNN through 1,2,5,10 and full connection of recurrent neurons in the TDRCNN. The results, represented in the Table 3.15, show that using recurrent neurons in small amount of all neurons (section 3.2.10 page 72), 13% - 33%, improves the accuracy of the network compared to the regular feed forward TDNN and full TDRCNN.

An example of the algorithm implementation area is the prediction of the temperature daily profile using the implemented BPTT stochastic update, mainly the Shuffle Update, due to the ability of the RCNN to use the history of previous steps stored inside. I already prepared the testing dataset for the temperature prediction, that consists of the measured temperature from the year 2000 till the end of February, 2011. The current experiments show that a deeper analysis is required to obtain results comparable with our FFNN.

The evaluation of the goals is:

1. Theoretical overview and analysis of RCNN

   Basic terms and theoretical overview of the work is introduced in the chapter 2 on the page 6. The specific implementation and proposed modifications with basic terminology is provided in the chapter 3 on the page 6.

2. Modification of the learning process using stochasticity

   The proposed stochastic modification of learning process is specified in the section 3.2.1, page 44 with 5 types of different approaches of the stochasticity usage.
3. Realize experiments focused on the introduced stochasticity

The chapter 3 on the page 34 contains proposed stochastic modification of the learning process as well as the realized experiments to prove the learning ability of modified algorithm. It is divided into three main parts: Daily Temperature Profile, modified implementation of Stochastic Weight Update on FFNN and modified implementation of Stochastic Weight Update on RCNN.

4. Evaluation of obtained results

The results of the experiments show that stochasticity applied in the learning process is comparable to the classical implementation of learning process. Complete evaluation of the experiment results for FFNN are in chapter 3.2.4 on the page 52 and for RCNN in chapter 3.2.11 on the page 78.

4.2 Future Work

In the future it will be meaningful to continue in the following tasks:

1. Find out a deterministic solution for the weight selection to be comparable to stochastic weight selection and provide deeper analyzes of the solution.

2. Introduce some general predictable weather model using stochastic approach for wider region, with several meteorological stations with aim to allow training from shorter history.
Chapter 5

Conclusion

This work was inspired by results of the Salvetti and Wilamowski (section 2.4.4 page 32) experiment which was focused on the introduction of stochasticity into the BP algorithm and provided experiments on the XOR problem. We have implemented five types of algorithms for introduction of stochasticity into BP described on Figure 3.7 page 46, that continuously involve more stochasticity on various parts of the BP algorithm and this stochasticity is increasing the algorithm types one by one. This stochasticity is focused on the learning part of the algorithm. The algorithm that we were mostly interested in is the Shuffle $\delta$Update (Figure 3.7 page 46). To prove that this algorithm can learn and can achieve better results than regular implementation, we started with a less difficult implementation of the BP algorithm - with the BP on the FFNN (see section 3.2.2 page 49).

The experiments focused on classification - circle-and-square (Figure 3.9) proves that the Shuffle $\delta$Update is able to learn and the results can be comparable to the regular BP algorithm implementation. The stochasticity provides the algorithm more resistance to the learning parameter $\gamma$ (section 3.2.4 page 52), which in the original BP algorithm has to be treated really carefully. The positive results with the experiments on FFNN (section 3.2.3 page 49) were the basis for the experiment provided on the RCNN using the BPTT algorithm. The results of the experiments (Figure 3.17, Table 3.12) on the RCNN proves the ability of the Shuffle $\delta$Update to be a comparable algorithm to the original BPTT.

The prediction of the Daily Temperature Profile with the RCNN (section 3.2.6 page 57) is quite difficult due to network complexity and the difficulty to find out the suitable parameters for the learning process of the RCNN(Figure 3.15, Table 3.11). To make it simpler, and find out the behavior of the Shuffle $\delta$ Update as well as the rest of the implemented algorithms, we provided experiments which show that the Shuffle $\delta$Update with a small amount of recurrent neurons (section 3.2.10 page 72), which also were selected stochastically, is more accurate than the original implementation.
The results of the experiments show that the Shuffle $\delta$Update is able to learn comparably to the original BPTT. It has a higher $\gamma$ resistance (section 3.2.9, page 68), which means that we can focus on the rest of the NN topology, and introducing stochastically chosen small amount of recurrent neurons (section 3.2.10, page 72), the algorithm can achieve a higher accuracy than the original BPTT. The same approach of the Shuffle $\delta$Update can be implemented for the BP algorithm on the FFNN (section 3.2.2 page 49) as well with comparable results on the RCNN (section 3.2.1 page 44). The stochasticity for some RCNN simplifies the implementation due to containing a pool of recurrent neurons that take part of the learning phase. The pool of neurons is the same in learning as in the execution phase of NN.

We believe that the proposed algorithm can be useful to simplify the implementation of backpropagation learning based for the RCNN.
<table>
<thead>
<tr>
<th>Shortcut</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NN</td>
<td>Neural Network</td>
</tr>
<tr>
<td>FFNN</td>
<td>Feed Forward Neural Network</td>
</tr>
<tr>
<td>RCNN</td>
<td>Recurrent Neural Network</td>
</tr>
<tr>
<td>TDNN</td>
<td>Time Delay Neural Network</td>
</tr>
<tr>
<td>TDRCNN</td>
<td>Time Delay Recurrent Neural Network</td>
</tr>
<tr>
<td>BP</td>
<td>Back Propagation</td>
</tr>
<tr>
<td>BPTT</td>
<td>Back Propagation Through Time</td>
</tr>
<tr>
<td>sBPTT</td>
<td>Stochastic weight selection of BPTT</td>
</tr>
<tr>
<td>MLP</td>
<td>Multilayer Perceptrons</td>
</tr>
<tr>
<td>TEKO</td>
<td>Heating company of Košice city</td>
</tr>
<tr>
<td>MSE</td>
<td>Mean Square Error</td>
</tr>
<tr>
<td>ART</td>
<td>Adaptive Resonance Theory / Cluster discovery networks</td>
</tr>
<tr>
<td>ESN</td>
<td>Echo State Networks</td>
</tr>
<tr>
<td>$i_{ni}$</td>
<td>Input into neuron $i$</td>
</tr>
<tr>
<td>DTP</td>
<td>Daily Temperature Profile</td>
</tr>
<tr>
<td>Phraseological collocation</td>
<td>meaning</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>train data</td>
<td>pattern of data for learning NN</td>
</tr>
<tr>
<td>test data</td>
<td>pattern of data which haven’t been presented to a NN during learning phase, mostly life data</td>
</tr>
<tr>
<td>learning</td>
<td>state of NN where the weights are changing and network is approximating function of training data</td>
</tr>
<tr>
<td>weight update</td>
<td>part of the learning process where the new information is set to the weight based on the error</td>
</tr>
<tr>
<td>weight modification</td>
<td>part of the learning process where the new information is set to the weight based on the error</td>
</tr>
<tr>
<td>error</td>
<td>different between required output and real output of a NN depends on the error function</td>
</tr>
<tr>
<td>weight selection</td>
<td>part of the learning process where the weight is selected to be updated</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>gamma parameter of NN is the gradient of learning, influences the speed of convergence</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>lambda parameter for steepness of the sigmoid activation function of neuron</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>alpha momentum is percentage amount of weight history influence</td>
</tr>
</tbody>
</table>
Bibliography


