Master in Artificial Intelligence (UPC-URV-UB)

Master of Science Thesis

NARX neural networks for sequence processing tasks

eng. Eugen Hristev

Advisor: prof. dr. René Alquézar Mancho

June 2012

Table of Contents

List of figures

List of tables

Abstract

This projects aims at researching and implementing a neural network architecture system for the NARX (Nonlinear AutoRegressive with eXogenous inputs) model, used in sequence processing tasks and particularly in time series prediction. The model can fallback to different types of architectures including time-delay neural networks and multi layer perceptron. The NARX simulator tests and compares the different architectures for both synthetic and real data, including the time series of BSE30 index, inflation rate and lake Huron water level. A guideline it's provided for any specialist in the fields of finance, weather forecasting, demography, sales, physics, etc. in order for him to be able to predict and analyze the forecast for any numerical based statistic.

1. Introduction

Sequence processing tasks and especially time series prediction have been a challenging subject for researchers, and especially for the ones using neural networks architectures. Usually a neural network can be used effectively for pattern classifying, but mainly for unstructured static data (not related by time constraint). However it is interesting to see how a neural network behaves on "*temporal pattern recognition*". The difference between the two types is that in the second case the patterns evolve in time, which makes recognition harder[1]. In consequence, the neural network has to adapt for these changes or at least consider them in the learning process.

Time series prediction is a perfect example of a temporal changing pattern. The time series is a sequence of discrete data taken every specific time interval, e.g. daily time series, or monthly, etc. Compared to a normal pattern that needs to be recognized, time series depend heavily on past values. A normal feed-forward network with a small number of inputs compared to the length of the time series can estimate the next value counting on little factors. In order to achieve a higher performance, like detecting long term dependencies, a new model has to be devised.

In order to be able to predict changing patterns, the neural network has to find a way to incorporate all previous values or at least sample them.

The Nonlinear AutoRegressive with eXogenous inputs model (NARX) is based on recurrent neural networks and aims to provide better long term dependencies detection in predicting tasks.

Time series are used in various domains like economy – exchange rates, or weather forecasting, or signal processing. Predicting such patterns is useful for everyone working in those fields: avoiding money loss in unexpected exchange rates changes, better weather prediction, or even better signal processing, by estimating what to expect on signal input.

This project aims to present and implement the NARX model, using neural networks, in a simulator and run experimental tests for measuring the performance of this model for time series prediction using both real world data and artificially generated data. The simulator is very flexible and allows fallback to both TDNN (time delay neural networks) and MLP (multilayer perceptron), both trained with standard backpropagation algorithm, in addition to NARX trained by BPTT (back propagation through time).

Although this work can be applied to any kind of sequence processing, it is focusing on time series prediction. One can use the simulator for a more general purpose, but the design is thought for time series, as they are the best example on which the sequence processing can be illustrated.

Section 2 presents some theory regarding time series and sequence processing task for a prediction system, along with a few functions and tests that can evaluate the quality of a prediction system. Section 3 presents the NARX model and how it can be implemented using neural networks. Further, Section 4 contains the architecture and implementation of the simulator, and Section 5 the results of the experiments and tests with the simulator. Finally Section 6 draws the conclusions and highlights the tracks for future development.

2. Time series prediction task

2.1. Time series concepts

Possible approaches in time series processing include autocorrelation, autoregression or residual research. Scientists have observed[2] that studying time series, some specific notions appear:

– *stationarity* – the property of a time series that the global joint distribution does not change over time – in consequence neither mean or variance;

– *nonstationarity* – the property of a time series that it's not stationary, usually it has a certain level of trend : linear, quadratic, etc.;

– *time series differencing* – the process of computing differences between pairs in order to try to make the series stationary;

– *trend terms* – the terms from the series that make it nonstationary, depending on the trend level, the series has to be differenced once or twice or more to become stationary;

– *random shock* – the random component of a time series, the shocks are reflected by the residuals;

– *residuals* – the errors that are visible after identifying the series model and trend if applicable;

– *lag* – the interval between two time steps, e.g. Y_t and Y_{t-1};

– *moving average terms* – the number of terms that describe the persistence of a random shock from one observation to the next, e.g. if it's 2, then every observation depends on 2 previous random shocks;

– *auto-regressive terms* - the number of terms in the model that describe the dependency among successive observations, e.g. a model with two auto-regressive terms is one in which an observation depends on two previous observations;

– *autocorrelation* – correlations between terms at different lags, values are repeating themselves in the series;

– *autocorrelation function* – the pattern of autocorrelation in numerous lags.

In order to correctly analyse the behaviour of the time series, one has to compute the mean, the variance, and see if the series is stationary. Further, if it's not stationary, it has to be differenced in order to find a stationary series, or logarithmed. After creating a model for the differenced series, the model can be generalised to the original series.

Subsequently, a neural network can be used to estimate the next values of the time series. Figure 1 presents a general predictor basic architecture[1].

In this architecture, one can notice that the observations of the time series are given as inputs, and the first step is the memory. The network has to memorize the input values for a sequence of last inputs, or with a different lag than the one from the original series, one with a lower granularity. Next, the system needs a predictor, that can base itself on both inputs and predicted values , but also on training data : correct outputs. In this case we denote x(t) as the input at time t, and y(t) the output. The memory usually has recurrent connections, while the predictor is a feedforward usual

network.

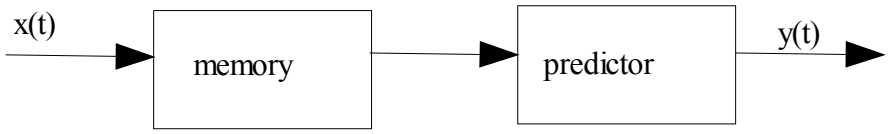

Fig. 1. A basic predictor architecture.

The design of such a system involves more steps[1]:

1) Decide the structure of the memory and predictor system: what type of neurons, activation functions, connections between them, number of neurons per layer.

2) Decide how the weights are adjusted. Usually given the inputs as a set of observations x(t) to x(t-τ), predictions are made, y(t) to y(t-τ), and the weights are adjusted such that the error to targets d(t) to d(t-τ) are minimised, usually in least squares sense.

3) Representation in memory : what should be kept and in what format. Memory should be domain dependant.

Memory representation has been studied in recent years and a number of models have been devised.

Tapped delay line memory is formed of *n* buffers containing most recent inputs. This memory is called tapped delay because every oldest input is replaced by a new one, and each input takes part in the learning process per turn. This model is the base for autoregressive methods. Figure 2 presents a simple tapped delay line architecture.

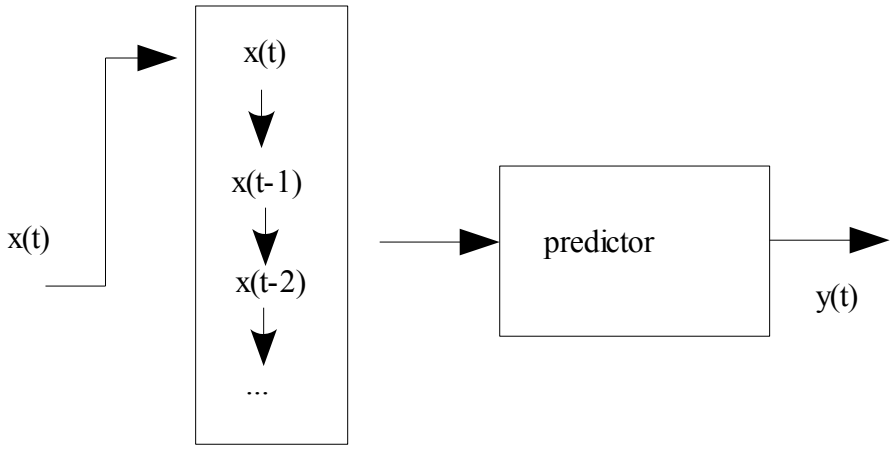

Fig. 2. A tapped delay line memory system

In order to comprehense more memory modules, we can denote the past values of the input with $\tilde{x}_i(t) = x(t - i)$; where *i* is the lag, and then treat $\tilde{x}$ as a convolution of the input sequence using a kernel function c_i.

Therefore,

$$\tilde{x}_i(t) = \sum_{\tau=1}^{t} c_i(t-\tau)\,x(\tau) \qquad (1)$$

For the tapped delay line, c_i will be

$$c_i(t) = \begin{cases} 1 \text{ if } t = i \\ 0 \text{ otherwise} \end{cases} \qquad (2)$$

Other kernel functions can imply different models of memory systems. For example, one can use a gaussian to sample values from the inputs, polynomial or exponential.

Consider two variables for a memory distribution, *depth* and *resolution*. Depth means how far away back in time the samples are taken from. Resolution will mean the lag value for which the samples are taken. Considering these, the linear distribution is low depth but high resolution. Opposed to that we have the exponential distribution, with a high depth but low resolution.

The exponential kernel function is

$c_i(t) = (1 - \mu_i)\ \mu^t_i$ where μ is an arbitrary constant in range [-1, 1].

One can observe that the samples with this distribution are more concentrated on the values close to the current input, and the lag increases with older and older entries. The exponential function does have the advantage of having theoretical values from all the inputs, even if the ones very far away are sampled at a very high lag.

One can combine the two variants into the *gamma memory[3]* and use advantages from both modules, with tweakable parameters. One can obtain a full family of gamma memories by varying the parameters from a tapped linear delay to a full exponential one.

The content of a memory is also very important. In previous examples, a linear memory mapping was done. Let's denote x' the new inputs, the ones after some mapping function is applied. In previous cases x'(t) is x(t). We can use a different mapping, such that the input values are differenced or any other preprocessing that would help us maintain data and extract the essential, like trends for example.

One can classify the memory content by the function that it's applied to the original inputs in order to obtain the mapped values. In some cases this function is the hidden layer neuron's activation function.

Sometimes in order to get the best out of the prediction system, the data needs to be preprocessed. This may include outlier removal or differentiation for example. Outlier removal means that the values that are very different from the pattern get removed : values that are probably caused by measurement errors or human errors, and may influence the prediction system.

Normalization has a very important role in preparing the data for using in a value-limited system, for example in a neural network using a sigmoid function : all the output values are in the range [0, 1].

By normalization for a series we understand creating a mapping to x'(t) in the following way: first compute the average of the input series $E(x)=\overline{x(t)}$, then compute the variance

$$Var(x)=\frac{\sum_{i=0}^{n}(x(i)-E(x))^2}{n}; \tag{3}$$

In this situation, the new series, the mapped ones will be:

$$x'(t)=\frac{x(t)-E(x)}{Var(x)}; \tag{4}$$

The new mapping will consist of both positive and negative values and it will be relative to the variance of the entire series. For the test data, one can use the same values for average and variance, the ones computed for the training data.

The memory can keep both the original data and the normalized (mapped) values. Further, the prediction system can work with both or only with the mapped values, having a mapping/demapping operation each time it has to interact with the user.

A memory is called adaptive if the memorised values are influenced by the learning process. This way, the hidden layer for example can connect back to the memory in order to recompute the stored values. The x' values will not be static as in a single function applied, but dynamic and will be influenced by the neurons in the network.

Types of neural networks algorithms for time series learning in recurrent networks include Back Propagation Through Time[4] or Real Time Recurrent Learning[5]. Both use adaptive memory as described above.

Having the three possible memory types (linear, exponential, gamma) and the two memory content classification, one can then split the memory into a taxonomy, and analyse which fits the best the sequence processing task.

The main goal for a network would be to adequately determine the function that converts the inputs x to the remapped x' inputs. Having weights on the inputs, denoted by *w*, the memory would have to try to determine *w* such that all the necessary information is kept. This includes any short term or long term dependencies between data, mostly the trend of the series and possible random shock.

A good approach would be gradient descent procedure. However in time series, gradient-descent has poor scalability, and it experiences the "*forgetness*" problem or the vanishing gradient problem[6]. This problem appears because expanding the Jacobian of the learning function over τ time steps tends towards zero when τ tends towards the beginning of the series. This means that the gradient descent learning procedure will value more the inputs closer to the current time rather than the old ones. In conclusion, the algorithm cannot discover and predict the so called long-term dependencies, that appear after a long time.

This projects aims to provide a solution to the problem of time series forecasting using a nonlinear model with neural networks that can detect long term dependencies. The purpose is also to experiment using generated data but also with real data, and determine which type of system is best suited for a specific time series prediction application.

2.2. Prediction evaluations

In order to evaluate the quality of a prediction, one can define a set of functions that can quantify the quality. Let's assume we have a time series *d(t)* and the prediction *y(t)* for a series of length *n*. The most simple function would be:

$$f_1 = \sum_{t=1}^{n} (y(t) - d(t))^2; \tag{5}$$

In this case f_1 is the sum of the squared deviations (SSE) of the predicted values compared to the target values. However f_1 is highly dependant on the series size: more terms to the series implies a bigger error, and this stops the possibility of comparing the quality of the prediction of two different series.

f_2 solves this problem by computing the mean:

$$f_2 = \sqrt{\frac{\sum_{t=1}^{n} (y(t) - d(t))^2}{n}}; \tag{6}$$

f_2 is the root-mean-square-error (RMSE). This function solves the problem of error cumulating, but it still has the problem of relative error. The computed mean error is only absolute, and highly depend on the series values. For example if we try to predict the values of the exchange rate and we have an f_2 value of 2.0, the prediction is very bad (ranging 30 % to 100 % for close value rates), but if we predict the stock markets values that range over one million per unit, an error of 2.0 per million is very small.

In order to solve this problem, f_3 will also divide the value by the average time series absolute values:

$$f_3 = \frac{\sqrt{\sum_{t=1}^{n}(y(t)-d(t))^2}}{\frac{\sqrt{n}\cdot\sum_{i=1}^{n}|d(i)|}{n}} = \frac{\sqrt{n\cdot\sum_{t=1}^{n}(y(t)-d(t))^2}}{\sum_{i=1}^{n}|d(i)|} = \frac{RMSE}{|\overline{d}|}; \quad (7)$$

f_3 is called the coefficient of variation of the RMSE, being RMSE divided by the mean $|\overline{d}|$. Although f_3 seems to cover the most problems, it only shows the mean error of the predicted values. If our prediction is off the scale for some certain values, then maybe the average itself will not be influenced a lot, but stay in a certain margin of error. The root mean squared based error calculations don't cover such cases when more penalties have to be included in the evaluation.

Further, we define f_4 as the normalized (by the target variance) RMSE.

$$f_4 = \sqrt{\frac{\sum_{t=1}^{n}(y(t)-d(t))^2}{\sum_{t=1}^{n}(d(t)-\overline{d})^2}}; \quad (8)$$

f_4 has an interesting property : if the predicted values are the exact average, the value of f_4 is 1. A better prediction than the average must have f_4 value below 1, and a worse prediction will have greater values.

Another interesting test would be a variant of *Random Walk:* compute the values of the test functions by using as prediction the previous value of the series. This would give us more insight about comparing the results: if the random walk tests results are similar with our prediction, then the prediction is no better than the random walk. For this purpose we define y_{RW} as the random walk prediction:

$$y_{RW}(t) = d(t-1); \quad (9)$$

Another problem is that the test functions if applied only on the training data do not cover the underfitting and overfitting problems. A good test should reveal if our model is too complicated for our data or contrary, too simple for our complex data pattern.

In order to evaluate a model, the Akaike Information Criteria[7] can be used, which is a simple and easy way to evaluate the fitness. However the AIC requires a precise determination of the degrees of freedom for a statistical model. Unfortunately neural networks create a model for the data but without an exact identification of the parameters. This would also mean that the degrees of freedom would be very hard to determine. There have been attempts to compute the degrees of freedom for the MLP[8], starting from the matrix of inputs and weights, by computing the eigenvalues, but this theory can only be used for a part of the NARX architecture, more less for the feedback loop. It has been proved though that neural networks can present a likelihood function by reduction to a conic model[9]. In this situation we can use the statistic tests that require the probability distribution and compare the prediction with the distribution of the real data.

We will use two important tests for "goodness of fit" : Kolmogorov – Smirnov[10] and Anderson – Darling[11], which require the distribution function of the data. Although these tests apply for continuous distributions of data, we will make some adaptation in order to apply for our case: consider the targets as part of a continuous distribution as well and compute the values for both the targets and predicted values. The comparison of the two can give an insight about whether they are part of the same distribution. These tests however do not tell us if a prediction is good or not, but they provide with a necessary condition: the points must be part of the same distribution. In order to have a good prediction, the ordering of the points is also important, and this is one point that the statistical tests for distribution match do not cover. We can say though that if the prediction and the

data is not from the same distribution, then the prediction is bad.

The common empirical cummulative data distribution function can be defined like the following:

$$F(t)=\frac{1}{n}\sum_{i=1}^{n}\mathbf{1}\{d(i)<t\};\tag{10}$$

where *n* is the number of data samples and *d* are the samples (the targets). **1** is the function that returns 1 is the condition is true or 0 otherwise. We can observe F is a probability so it will be in the range [0, 1].

F can then be used in the tests.

For the both tests, the result of the test has to be compared with a critical value specific to the target distribution. If the computed statistic is greater then the critical value, then we can reject the null hypothesis that the samples are from the same distribution, and accept the alternative, that they are from a different distribution. The critical values are standard for certain distributions like the *normal distribution*. Because the distribution of the targets – time series is unknown, this value has to be determined empirically.

The Kolmogorov – Smirnov test (abbreviated KS) has the advantage of being independent of the actual values of the series and distribution, so the statistic itself is absolute. Another important advantage is being exact: compared to root mean square tests or chi-squared tests, KS does not require an adequate size of data, it can be computed with decent results for any size of data. A disadvantage for KS is that it favours mostly the center of the data and leaves the edges with less influence on the result.

The KS statistic is defined as:

$$f_{KS}=\max_{i}|F_1(i)-F_2(i)|;\tag{11}$$

where F_1 and F_2 are the empirical distribution functions for the two series.

The formula looks pretty simple and it basically finds the largest difference in absolute value between the cummulated distribution functions: the point where the cummulated error reaches the maximum.

The Anderson – Darling test (abbreviated AD) starts from KS but it has a better coverage of the data outside the center : the head and the tail of the series.

$$f_{AD}=-N-\sum_{i=1}^{N}\frac{2\cdot i-1}{N}[\ln F(y_i)+\ln F(1-y_{N+1-i})];\tag{12}$$

In this case the *y* are the predictions and *F* is the distribution of the targets. This adaptation is required because the original distribution is unknown, but we have it's cummulated distribution function. Another point that has to be taken into consideration: it is possible that the distribution function of the targets may return a value of zero if the predicted value is below any target value. This is not possible for the original formula because for every value there is at least one that matches (the actual value we want to compute for) so the probability never reaches zero. To avoid logarithm from zero, we must ignore the specific cases when the cummulative distribution function returns 0.

In conclusion, the analysis of a distribution match would require the computation of the above functions and tests and compare them.

The statistics test together with RMS functions and their variants can give us a certain amount of information that we would require in order to say if a prediction is better or worse than another.

The next section presents the NARX model, which starts from the gradient descent problem, with a nonlinear memory and tries to discover long-term dependencies in input data.

3. The NARX model

3.1. Theoretical NARX and neural networks

Nonlinear AutoRegressive with eXogenous inputs[12] model is a type of recurrent neural network defined by the following:

$$y(t) = f(x(t), \ldots, x(t-a), y(t-1), \ldots, y(t-b), d(t-1), \ldots, d(t-b)); \quad (13)$$

where *d* are the targets for the time series that we want to predict, *y* are the past predicted values by the model, *a, b* are the input and output order, *x* are the exogenous variables and *f* is a nonlinear function.

The model purpose is to predict the next value of the time series taking into account other time series that influence our own, but also past values of the series or past predictions.

In the model we can observe the exogenous variables: variables that influence the value of our time series, the one we want to predict. The input order gives the number of past exogenous variables that are fed into the system. We can also note that the exogenous variables have an arbitrary number (can be none, one, or more). In general, the exogenous variables are time series as well. We can use the exogenous variables values starting from current time *t* until *t – a*, where a is the input order. The input variables among with their order are called the *input regressor*.

The *y* are the past predicted values. Because we want to predict the value at the current time *t*, we can use values starting from *t – 1* to *t – b*, where *b* is the output order – the number of past predictions fed into the model. These output values among with their order are called the *output regressor*.

The targets *d* represent the real values of the time series that we want to predict, which are also fed into the system. The same order as for past predicted values is used. In case these values are missing, the system will try to predict the next values of the time series only from the exogenous variables or using the feedback from past predicted values.

The first system parameter is the number of inputs (exogenous) *M*. This means that the NARX can fall back to a prediction system without exogenous inputs if needed, in which case *M* is 0. The input order (how many previous values are given to the system) is denoted by *a*. Another parameter is the actual order of delayed outputs or delayed targets. This is denoted by *b*. The parameter *i* represents a certain exogenous variable taken into account. In this way, x(*t)* actually represents a vector of exogenous variables at the time *t,* with *i* varying from 1 to *M*.

The outputs *y* actually represent a vector of outputs, so we denote the number of output variables as *N*.

The system parameters can be resumed in the following:

Inputs (exogenous) : *M ;*

Outputs (predicted): *N* ;

Input delays order : *a* ; can be > 0 only if *M > 0*

Output delays order: *b* ;

Varying the system parameters can result into different types of architecture, with or without exogenous inputs, with or without delayed outputs (feedback), etc., resulting into simpler models for specific purposes.

With the above notations, the output of the network for the time *t*, the prediction, is *y(t)*, and because we have the targets *d(t)* we can compute the error *e(t)* as *d(t) – y(t)*.

In Figure 3, one can observe the complete architecture.

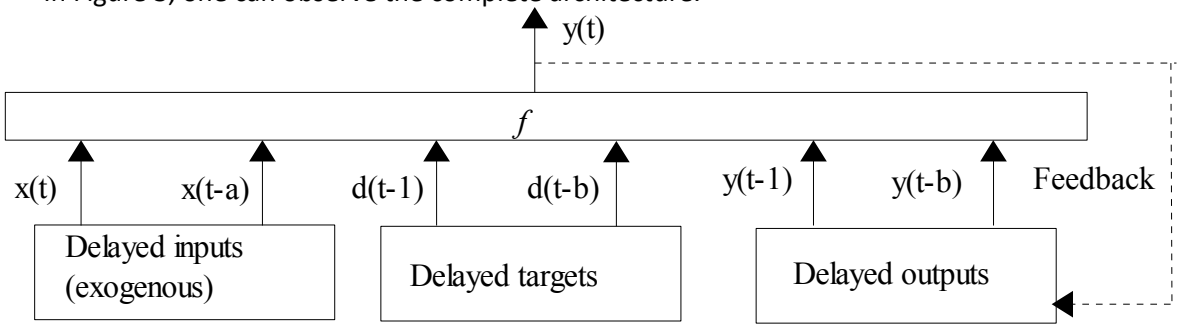

Fig. 3. NARX model

NARX model can then be built on a recurrent neural network, trained by BPTT (backpropagation through time) algorithm or simple BP (back propagation) if the feedback is removed, in this case resulting an MLP (multilayer perceptron).

In Figure 4, the NARX model using neural networks is presented.

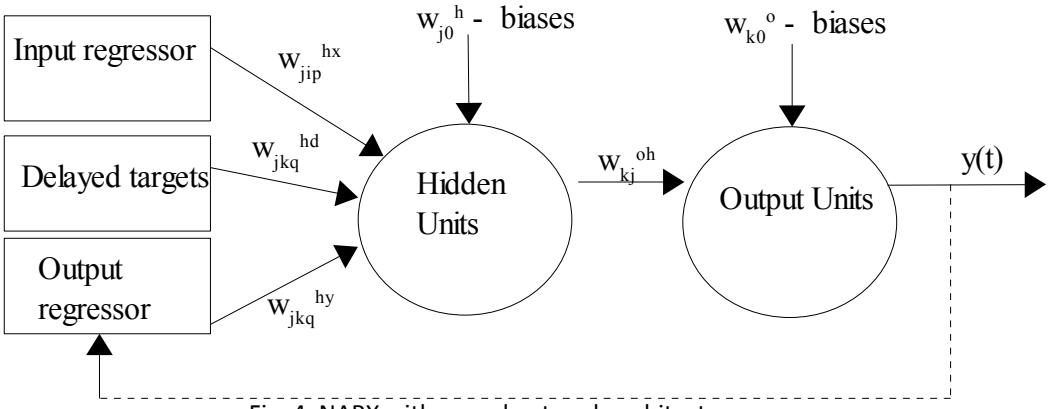

Fig. 4. NARX with neural network architecture.

The neural network holds for the nonlinear function *f*, modelled this way[13]. The learning algorithm for the neural network can be gradient descent, with the addition from BPTT for the feedback into the output regressor.

The neural network adds new parameters to the architecture: *H* the number of hidden units. Because we may have a multiple output, the number of output units is *N*.

Another option that has to be considered for the neural network is the selection of the activation functions. The hidden units can have a nonlinear limited function like the logistic sigmoid :

$$sigmoid(x) = \frac{1}{1+e^{-x}}; \tag{14}$$

with derivative :

$$sigmoid'(x) = sigmoid(x)[1 - sigmoid(x)]; \tag{15}$$

which is good in general if the inputs are in a range that is not affected by the limits: the sigmoid falls to 1 for large values and to 0 for large negative values. This problem can affect the performance for these type of numbers, if not taken into consideration. To avoid this problem, one can use data preprocessing like normalizing , or modifying the respective activation function.

The B sigmoid tries to solve this problem by adding an extra parameter which will alter the slope of the function:

$$sigmoid(x,\beta)=\frac{1}{1+e^{-\beta x}}; \quad (16)$$

with derivative:

$$sigmoid'(x,\beta)=\beta\, sigmoid(x,\beta)[1-sigmoid(x,\beta)]; \quad (17)$$

This function however does not solve the initial problem of the sigmoid. We can use the antisymmetric logarithm function, which is similar with sigmoid but does not have the $+\infty$ and $-\infty$ asymptots:

$$antisymlog(x)=sgn(x)\log(1+|x|); \quad (18)$$

with derivative:

$$antisymlog'(x)=\frac{1}{1+|x|}; \quad (19)$$

For the output layer units, the linear identity function is applied:

$$f(x)=x; \quad (20)$$

because the output units must produce the output by summing the output from all the hidden units, if this activation function is different, some outputs cannot happen (e.g. above 1.0 for sigmoid).

We denote the neural network weights in the following way:

w_{jip}^{hx} – the weights for the exogenous inputs *x*, connecting the *j* hidden unit (*h*) with the *p* input delay for the i^{th} exogenous variable.

w_{jkq}^{hd} – the weights for the targets *d*, connecting the *j* hidden unit (*h*) with the q^{th} delayed target for the k^{th} target variable.

w_{jkq}^{hy} – the weights for the feedback outputs *y*, connecting the *j* hidden unit (*h*) with the q^{th} delayed output for the k^{th} output variable.

w_{kj}^{oh} – the weights for connections between the k^{th} output unit and the j^{th} hidden unit respectively.

w_{j0}^{h} – the bias for the *j* hidden unit (*h*).

w_{k0}^{o} – the bias for the *k* output unit (*o*).

In addition to the parameters defined above, we use the following indexes to address a specific variable or unit:

Outputs *k* : $1 \leqslant k \leqslant N$;
Hidden units *j* : $1 \leqslant j \leqslant H$;
Inputs (exogenous) *i* : $1 \leqslant i \leqslant M$;
Input delay *p*: $0 \leqslant p \leqslant a$;
Output delay *q*: $1 \leqslant q \leqslant b$;

Having the NARX specified in this way, one can ask if the vanishing gradient problem still exists, because we have a similar learning algorithm with the Multi-Layer Perceptron (MLP), and a very similar network architecture. The answer is yes, the vanishing gradient still exists, however, the model behaves much better then the other types, because of the output memories represent jump-ahead connections in the time folded network[8]. What this means is that if we take the network and represent multiple instances of it, basically each instance at a moment of time connects to another one far in the future. This improves the long term dependency detection and somehow softens the vanishing gradient problem that the other type of networks have. Such behavior is also seen in BPTT.

The NARX network can use the regressors with different lags[8], this time they no longer have the same lag as the series, but having τ now. This means that the input regressor is actually a τ-separated selection of inputs from the original time series. This allows the network to cover more space in order to determine long term dependencies, but with bigger granularity. In conclusion the

regressors will adjust for the trend of the series, in long term. However one can keep the same lag as the original series to the output regressor for example, and it will adjust to smaller changes in the time series, included in the random shock.

This type of NARX network has proven to be a successful alternative to TDNN for a lot of practical examples, including chaotic laser pulsations or video traffic time series[8], where it behaved significantly better.

One can observe that the regressors actually provide the memory for the network, either a linear tapped-delay as explained in previous section, or an adaptive memory if the lag changes from 1 to τ for better series coverage.

It has been proved that a NARX network behaves better on long term dependencies by using a single neuron network compared to a recurrent network trained by BPTT[6]. In the case of NARX, the Jacobian component of the gradient is emphasized from terms farther in the past. This means that the fraction of the Jacobian value given by older values is higher for this type of network. In a regular network case, the Jacobian would decay faster, so the old values are practically insignificant to the later steps of the prediction.

NARX networks have been tested for finite automata prediction[6]. In this scenario, input strings were given to the network, and in a specific case where a certain character was found in a position or not, the string was accepted or rejected. 500 random strings were created and tests have been done with both Elman and NARX networks. NARX behaved much better, with around 12 input delays, the rate of correct prediction was nearly 100 %, which was nearly the same for Elman with 6 input delays, but the performance rapidly degraded for Elman as the input delays number increased, while for NARX the performance degraded softer.

An interesting fact is that NARX networks have been proved to be Turing equivalent[7]. This means that any problem can be modeled and solved using a NARX network. In theory, a NARX network can replace any recurrent network that is currently being used for a problem, without any computational power loss.

NARX model has proved to be effective for other types of chaotic series as well[14], including the chaotic Mackey-Glass series, and Fractal Weierstrass series. For these series, a good prediction has been found with an average input order and low output order. The model has predicted values with 99 % accuracy, comparing them with the original values.

For the BET-stock market (Bucharest stock market), NARX has proven again to have good results, with the Levenberg – Marquardt optimization and Bayesian regularization. In this scenario, the accuracy was 96 %.

3.2. NARX variants

Having the system parameters, one can find older models that are covered by NARX, but also subsystems of NARX.

1) MLP (multi-layer perceptron with exogenous variables). NARX model can fall back to MLP if the parameters are M > 0, a = 0, b = 0. Having *a* zero means no delayed exogenous inputs are given to the system, but because M > 0, we have exogenous variables at the current time index. Having *b* zero means no feedback loop nor delayed targets. In this scenario the prediction is done using only exogenous values. This architecture is in fact the multi-layer perceptron.

2) TDNN – X (with exogenous variables only). Parameters are M > 0, a > 0, b = 0. This time the exogenous variables are present with delay as well, but no feedback. This architecture is the Time-Delay Neural Network X (from eXogenous). In this scenario only one block is present – the

exogenous variables.

3) TDNN – D = NAR – D (without exogenous variables but with delayed targets). Parameters are M = 0, a = 0, b > 0. In this case there are no exogenous variables but we have the delayed targets. That is why the model has the D suffix (from the targets *d*). Even though *b* is > 0, only the targets are present: there is no connection between the *y* block and the system: $\exists\ w_{jkq}^{hd}$ but $\nexists\ w_{jkq}^{hy}$.

4) NARX – D (with exogenous variables and delayed targets). Parameters are M > 0, a > 0, b > 0. In this case both exogenous variables and the delayed targets are present. The only block missing is the feedback. $\nexists\ w_{jkq}^{hy}$, $\exists\ w_{jkq}^{hd}$, $\exists\ w_{jip}^{hx}$.

5) NARX – Y (with exogenous variables and delayed feedback outputs). Parameters are again M > 0, a > 0, b > 0. This time there are both exogenous variables and delayed feedback outputs, but no delayed targets. $\exists\ w_{jkq}^{hy}$, $\nexists\ w_{jkq}^{hd}$, $\exists\ w_{jip}^{hx}$.

6) NARX – DY (with all inputs). M > 0 , a > 0, b > 0, $\exists\ w_{jkq}^{hy}$, $\exists\ w_{jkq}^{hd}$, $\exists\ w_{jip}^{hx}$.

7) NAR – Y (no exogenous and no delayed targets, just prediction feedback). M = 0, a = 0, b > 0, $\exists\ w_{jkq}^{hy}$, $\nexists\ w_{jkq}^{hd}$, $\nexists\ w_{jip}^{hx}$.

8) NAR – DY (no exogenous but delayed targets and prediction feedback). M = 0, a = 0, b > 0, $\exists\ w_{jkq}^{hy}$, $\exists\ w_{jkq}^{hd}$, $\nexists\ w_{jip}^{hx}$.

Out of all the possible variants, MLP, TDNN – X, TDNN – D (NAR – D) and NARX – D can be trained with simple backpropagation. NARX – Y , NARX – DY, NAR – Y and NAR – DY must be trained with BPTT.

We can observe that the models that don't have the X in their name (TDNN – D , NAR – Y and NAR – DY) don't use exogenous variables, so the time series forecasting is simple. The other models include exogenous variables.

3.3. Backpropagation through time

In the NARX model we can observe that we have a feedback loop from the output of the network back to the input. This special connection is part of recurrent neural networks, and cannot be trained with simple backpropagation algorithm. For this, the BPTT (backpropagation through time) algorithm needs to be used.

The BPTT algorithm for processing a training sequence of length *t* is presented next:

01: Unfold the network in time for *t* necessary steps.
02: For each *i* instance of the network:
    021: Add the output of output units of the network instance *i-1* to the input of network instance *i* and the *i* input values from the inputs
    022: Compute the output for current instance
    023: Propagate the output further
03: For the final instance *t*, compute the error as *e(t) = d(t) – y(t)*
04: For each instance *i* of the network in reverse order:
    041: Propagate the error back using backpropagation.
    042: Aggregate the propagated error with the current error *e(i) = d(i) – y(i)* at time step *i*.
    043: Adjust weights (and biases).
05: Fold the network back by averaging all the *t* weight instances (and biases) into a single instance of weights (and bias).
06: End.

We can observe that the algorithm for each training sequence (at each epoch) unfolds the network in time as much as the length of the input series is. This can bring a penalty of space, but there are some enhancements that can be done: don't keep separate copies, but just keep the weights; or recompute using a single instance because every instance is the same.

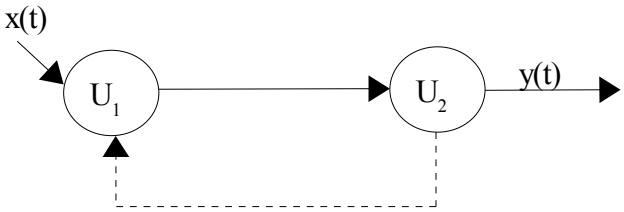

Fig. 5. A recurrent neural network.

The unfolding in time is depicted in Figure 5 and Figure 6. In Figure 5 we can see the original recurrent neural network, and in Figure 6 the network unfolded through time for 3 time steps.

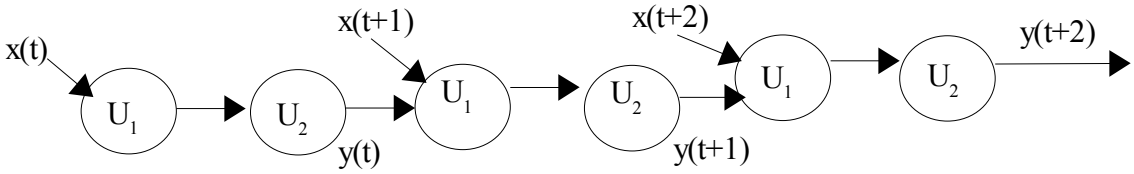

Fig. 6. The network in Figure 5 unfolded through time.

We can observe that a single past output of the network is being sent to the next instance of the network. If more are being sent (e.g. for 2, each network will send the output to the next and the following instances), then the *order* will be 2. This is the significance of the *output regressor* in the NARX model. The unfolding needs to be done for *n* instances, if the input series has the size *n*.

An important difference is the behaviour of the output units. In normal backpropagation, the output units only compute their error by calculating a difference from the targets, while in BPTT, besides this computation, every output unit receives a delta from the hidden units of the following instance, so that the error can be reduced not only with respect to outputs, but also to the derivative of the error coming from the following instances.

The time complexity for the algorithm is $O(N^2)$ per time step, and the space complexity $O(N^2 + n \cdot \{N+M\})$ where N is the number of units in one instance, M the number of exogenous variables (x) and n the length of the time series.

BPTT must be applied to the learning steps of all NARX variants with output regressor (Y).

4. Application architecture and implementation details

The application is a NARX simulator, that can also fall back to any of the described architectures in Section 3.2. The simulator is written in C++ and uses the Qt[1] graphical library for the interface. The tests can be run on both generated series and on real data. Figure 7 presents the architecture. Requirements to run the application are: 32-bit Windows based system with MS VC++ redistributables installed and .NET framework 3.5.

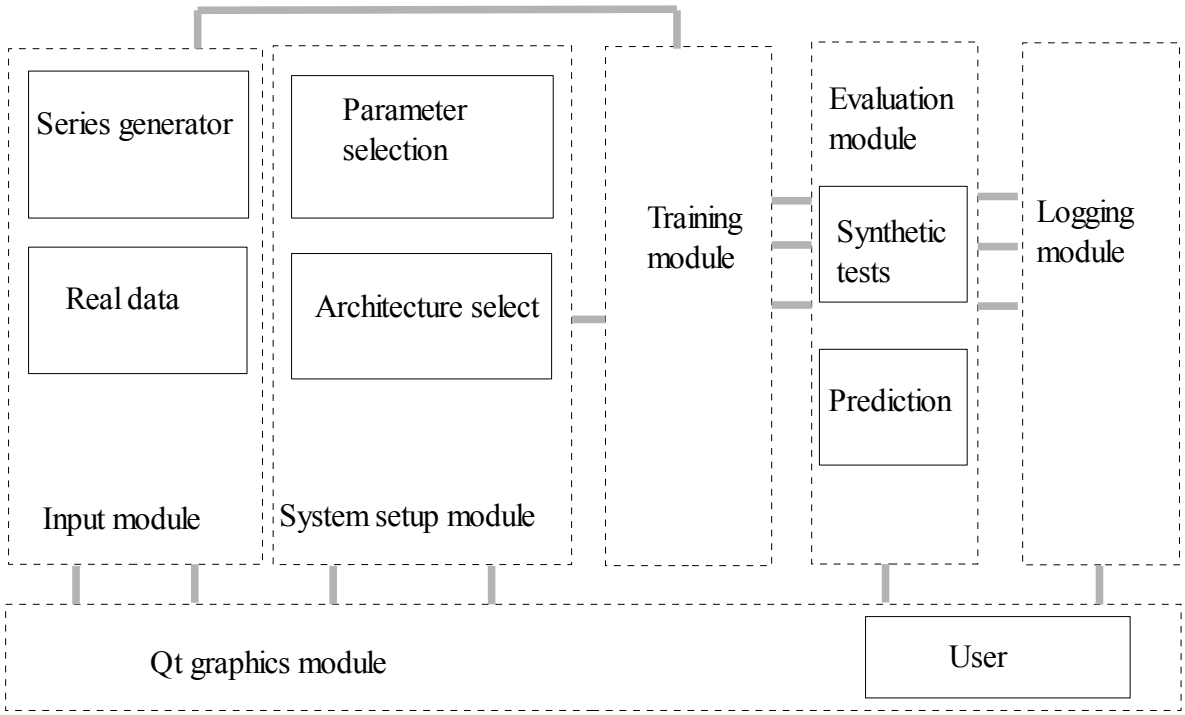

Fig. 7. The simulation engine architecture.

4.1. Input module

The input module consists in generating a time series or loading an existing one from file. The series generator has the following parameters:

– start value : a real number representing the first value of the exogenous variable used for generating the series;

– end value : a real number representing the end value for the exogenous variable;

– series length : an integer number representing the number of values in the series;

– base function : the basic function that will be applied to the exogenous value in order to

1 http://qt.nokia.com

generate the series value; can be one of *sinus*, *normal logarithm, normal exponential, square*, *hiperbolic* (*1/x*), *linear* (*f(x) = x*);

– noise factor : representing a percent of the final value that is added or substracted from the value in order to artificially create a noise: a random value between - noise % and + noise % is generated and then added to the series value.

After generating the series, the user can select whether to use the exogenous variable into the model or not.

This generator does not support usage of multiple exogenous variables or having a recurrent dependency (of previous values of the series). However another option is available – using a predefined function for a series: the generator has a few built-in series with formulas that can be used for testing.

The second part of the input module is represented by the series loader. The series must be in a plain text file having the following format: an integer number representing the series length: *n*, then the number of exogenous values used: *M*, then the number of targets *N*, and then for each exogenous variable, *n* real numbers representing the exogenous values for this variable for each time step. In the end the file must contain another *n* values for each actual *N* time series values. This part of the input module allows the simulator to train and predict real life values (not artificially generated ones).

For all the data, real or artificial, the user can select whether to normalize the values as described in Section 2.1.

4.2. System setup module

After the series have been generated, the user can pick which variables will be used further in the model for the training and prediction.

The parameter selection module allows the user to pick the alpha coefficient of learning : the number that balance between exploration and exploitation. If alpha is too low, then the network will be inclined more towards exploitation, because the error will be propagated less towards adjusting the weights, thus the learning process can be slow (many epochs required), but it's not so sensitive to major changes in patterns. In this way if some patterns are completely different from the previous ones, the network will not adjust very fast to them but keep in memory what it learned previously. Opposite to that is the exploration: if alpha is set too high, the network will respond very fast to changes, and may "forget" previous learned patterns. Learning may be faster with a higher alpha but with less precision and it can become sensitive to outliers. An usual value for alpha is 0.2.

Another parameter that can be set is the number of epochs : the number of iterations through all the series that the network will perform in the training process. An usual epoch number is 500 – 1000.

Next step is the selection of activation functions for hidden and output units. Possible choises are *sigmoid, β-sigmoid, antisymmetric logarithm* and *linear.*

The architecture select module represents the actual pick of the type of NARX used: user can choose whether to use exogenous variables, delayed outputs or targets. Depending on these choices, the system will identify one of the architectures presented in Section 3.2. Also in this module the user can select the number of hidden units that the neural network will have. More hidden units will imply a longer time for training, but may provide better results. However a too large number of hidden units will lead to overfitting, in which case the model will not have a good prediction due to too high complexity. Having a too small number for the hidden units can cause underfitting, a model too simple for the function that we want to predict. The number of hidden units has to be tested and

adjusted for each prediction, for each time series, according to evaluation (either by tests or manually).

Further, according to the type of architecture selected, the user is pointed to input the desired input and output orders.

4.3. Training module

The training module is the most important in the architecture, as it implements all the required learning algorithms: backpropagation and backpropagation through time, and it holds the architecture of the entire NARX system. In Figure 8 we can observe a detailed architecture for this specific module.

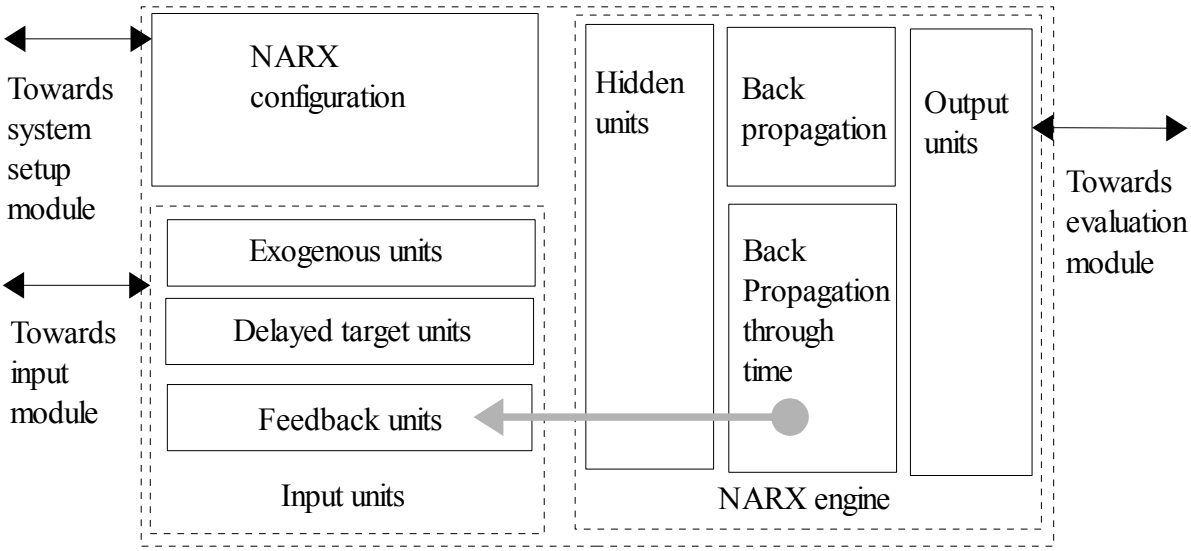

Fig. 8. Training module architecture.

We can see how the module interacts with the other different modules. The NARX configuration influences what types of units will be active during the training and prediction of the system. The input module will feed the data for the input units and for the targets for error computing.

The training module is composed in a similar way with the NARX theoretical architecture: input units, which consist of exogenous units, delayed target units and feedback units. The activation of each depends on the selected architecture. Further, the hidden units are present, a single layer, and the output units. One of the two algorithms is used for training: for non feedback types, simple backpropagation, while for feedback types, the backpropagation through time. In the end, the output units feed the values to the evaluation module or back to feedback units.

For the simple BP, after each value from the time series, the weights are adjusted: the errors are computed for output units and hidden units, and after that the weights are adjusted accordingly. A slight difference is for the BPTT: in order to avoid too much memory usage, the simulator does not make a separate copy of the architecture for each element of the time series, but only three copies are used.

Initially a first copy computes the forward way, and all the outputs are computed. For example, at time step *t*, into the architecture are fed the *b* previous computed values for output, along with the *a* exogenous values and the b delayed targets if necesarry. The inputs are saved in a specific

feedback info structure.

For the back way, the previous saved outputs and inputs are reapplied into the network and the errors are computed. Weights are adjusted and a separate copy of the whole NARX cummulates all the weights (for later averaging when the network collapses back).

The output units provide the prediction results necesarry for evaluation. In consequence the values are being sent to the evaluation module.

Figure 9 presents the C++ class scheme for this module, generated using Microsoft Visual Studio 2010.

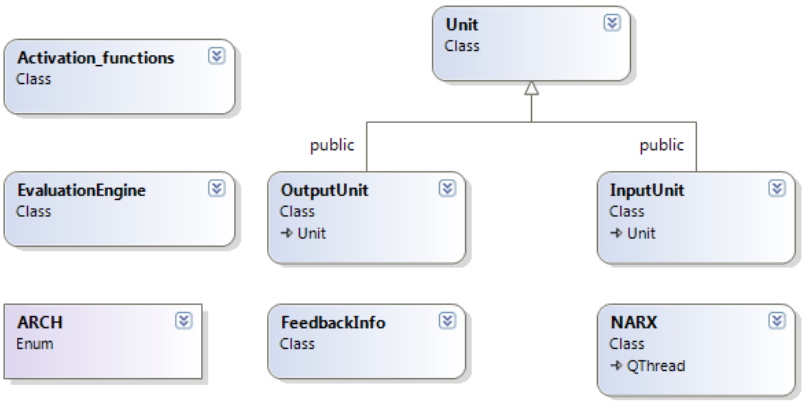

Fig. 9. Class inheritance diagram

We can see that a basic class for unit is the class Unit, which is superseeded by InputUnit and OutputUnit. Basic Unit class stands for hidden units only. The NARX class is in fact a QThread, a specific implementation of threads offered by Qt library, it's purpose is to do the training in a different thread, behind the GUI so the application does not freeze during the process. Training can happen in a very short time or in a very long time, depending on the size of the series but also on the complexity of the architecture.

More deep, the interconnections between the classes are put together so that the engine works: every Unit has an Activation function, the NARX class has an architecture of type ARCH, etc.

4.4. Evaluation module

The evaluation module consists of two big parts: the synthetic tests, which are the implementation of the test functions described in Section 2.2, and the prediction module which allows the user to input new values for the exogenous variables (that were not used in the training) in order to see how the network predicts new values for the next time steps.

The synthetic tests consist in the implementation of the functions f_1 to f_4, together with the distribution tests (Kolmogorov – Smirnov and Darlington – Anderson). Also the Random Walk tests are computed : the same functions , but the previous value is used as prediction.

The prediction module interacts with the user for getting new inputs for the exogenous series used in the training, and the predicted values are presented, for as many time steps as necesarry.

The evaluation module's purpose is to compare and realize how good the prediction is, according to all the indicators that have been presented in Section 2. In consequence the desired targets must be supplied as well. The user can then compare different architectures and tweak the parameters in order to obtain a better prediction for a specific problem.

4.5. Logging module

The logging module offers two types of log from inside the application code: either show the log to the log window in the GUI, or to file. The GUI shows minimal information about the system: how many targets were found, how many exogenous variables loaded, or how the normalization of the series was performed: what is the average and variance of the specific series that was treated.

Because the GUI updates are done asynchronously (using the signal and slot mechanism from Qt) and the update is very time consuming, the file logging comes in to solve this problem: more detailed information like every epoch result are printed to file.

4.6. Qt graphics module

The graphic module is created for direct interaction with the user: loading series, performing architecture select, training and prediction. The GUI is intuitive and easy to use, using a tab format for each step of the program. The Annexes presents several screenshots of the application during it's run.

5. Experimental results and performance evaluation

5.1. Basic functions

The most basic test for the predictor system is a basic function. The basic functions are generated artificially using the generator module. Tests were realized using a series of 500 elements with sinus function and a noise factor of 5 %. Out of the 500, 10 % were used for test and 90 % for training. The average of the series is E = 0.459094 and variance Var = 0.0634804.

Table 1 presents a sample of the data.

Table 1

Basic sinus test function

Index	Exogenous value (X)	Series value (D)
1	0.000	0
2	0.002	0.00204
3	0.004	0.0040399
...	...	...
52	0.102	0.0967321
...	...	...
447	0.890	0.784842
...	...	...
500	0.998	0.840389

Table 2

MLP training results

Epoch no	f_1 SSE	f_2 RMSE	f_3 COEF_VAR	f_4 NRMSE	KS	DA
1	2.55166	0.0753017	0.180364	0.329309	0.04	-29.6103
2	1.42454	0.056264	0.134764	0.246053	0.0377778	-25.109
3	1.41628	0.0561007	0.134373	0.245339	0.0377778	-25.0759
4	1.40729	0.0559224	0.133946	0.244559	0.0377778	-25.0534
...	...	...	...	...	...	...
99	0.0961644	0.0146184	0.0146184	0.0639292	0.0155556	-8.0822
100	0.0961567	0.0146178	0.0350129	0.0639266	0.0155556	-8.0822

Table 2 presents basic results from the training period, using the MLP architecture. The learning rate was 0.2 and the number of epochs 100. The number of hidden units was 3. Activation functions were sigmoid for hidden units and linear for output units.

We can observe that the absolute squared errors drop in time, for each epoch. Values for the RMSE (f_2) drop significantly. Also we can see that the coefficient of variation of the RMSE (f_3) is small. NRMSE (f_4) indicates that the prediction given by the system is significantly better than the average prediction (predicting the average value of the series all the time). The distribution match tests (KS and DA) also suggest that the values are more and more close to the same distribution as the learning process advances.

In Table 3 we can see the values for test of the MLP – after each training epoch, a run through the test values was performed in order to see how the results change with the training.

Table 3

MLP test results

Epoch no	f_1 SSE	f_2 RMSE	f_3 COEF_VAR	f_4 NRMSE	KS	DA
1	0.0547032	0.0330767	0.0408861	1.39083	0.8	37.1533
2	0.0543738	0.0329769	0.0407628	1.38663	0.8	37.1533
...	...	...	...	...	...	...
99	0.0261344	0.0228624	0.0284041	0.78307	0.14	-10.3138
100	0.0261331	0.0228618	0.0284034	0.783051	0.14	-10.3138

We can see that the MLP behaves fairly well on the test set, with very low sum of squares errors and with a better than average prediction as indicated by the NRMSE.

The results on the training set are better, indicating that the network has learned the training sequence.

The distribution tests indicate that the points are from the same distribution, with about the same values for both KS and DA tests , for training and for test sets.

The MLP is a simple network, capable of learning a simple function, but it has no sense of order. If the values in the training set are put in a random order, the learning procedure and the results would be the same.

In order to see if the order makes any difference for the prediction, the NARX-D architecture will add the delayed targets that can make the network learn the order. Table 4 presents the results of the same function, applied on a NARX-D architecture, with input regressor 0 (only exogenous variables at the current time) and with output regressor 2 (previous two target values).

Table 4

NARX-D training and test results

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
198	0.173298	0.0857101	0.0260221	0.751744
199	0.172608	0.0855394	0.02598	0.751136
200	0.171929	0.0853708	0.0259384	0.750534

We can see that adding the delayed targets to the architecture, the performance improves. Having the two previous target values added as inputs to the architecture makes the system aware of previous values, this will make the network learn about the trend of the function, and makes the order of the values important. Compared to the MLP, if we reorder the values, the prediction would change significantly.

However, the exogenous values are very important for the prediction, if we remove them and we only use the NAR-D architecture, the results are worse. We can see the results in table 5.

Table 5

NAR-D training and test results

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
99	0.14033	0.0771006	0.0370714	1.04475
100	0.140404	0.077119	0.037071	1.04474

Even though the NAR-D learns quickly enough, and the sum of squares errors are low, the normalized RMSE shows us that the prediction is worse than the average. The results indicate that the exogenous variable play a very important role in prediction. The training results are poorer as well, which suggests that the network did not fit on the training set as good as the previous two architectures.

Next, it is interesting to see if the network is capable of learning in a better or worse way the same series, but with delayed exogenous variables. The input regressor would give important order information to the network, not directly through targets, but from the exogenous variable that generate the series. Table 6 presents the results of the TDNN-X architecture, applied with a learning rate of 0.1 over 400 epochs, and with an input regressor of order 2.

Table 6

TDNN-X training and test results

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
98	0.092943	0.063301	0.0238061	0.793357
99	0.092936	0.0632986	0.0238068	0.793369
100	0.092929	0.0632962	0.0238074	0.79338

The results indicate that there is no improvement over the regular MLP. This is due to the cause that the function that generates the time series does not take into account previous values of the exogenous variable (is not recurrent with respect to this variable), so the network cannot find any longer dependency except the one with the current exogenous time index.

An interesting fact happens on TDNN-X test. Even though the errors for the training set decrease in time , the errors for the test set increase. This suggest that an overfitting has happened: the network tries too much to adjust on the training set, overfitting on it, while the test set errors grow in time.

In conclusion, the basic function is approximated best by a simple MLP, with small improvement if we take into account the previous target values for the order. A more complicated variant, using TDNN, does not bring any performance, while taking into account only previous targets makes the network unable to achieve the same result.

5.2. Recurrent functions

A more advanced type of function, with a sequence (temporal) dependency is required in order to see the difference that the regressors bring to the architecture. The simple functions do not behave too much different (better) for other types of architectures from the model.

The first proposed predefined series that is artificially generated is the following:

$$d_1(t)=\sin(x(t)+y(t)\cdot d_1(t-1))\cdot z(t)+\tan(d_1(t-2)-d_1(t-3)); \quad (21)$$

This series has the property that it uses previous generated values. We can see that the previous three values are used, for time *t-1*, *t-2* and *t-3*.

The series average and variance are the following E = 6.07627, Var = 6046.59. Table 7 shows a sample of the series before normalization.

Table 7

Recurrent test function d_1

Index	Exogenous value (X)	Exogenous value (Y)	Exogenous value (Z)	Series value (D)
1	0.000	3	1.1	0.175878
2	0.002	3.05	1.07	0.526734
3	0.004	3.1	1.04	1.23975
...	...	...	...	...
52	0.102	5.5	-0.4	0.408732
...	...	...	...	...
447	0.890	25.25	-12.25	-13.9306
...	...	...	...	...
500	0.998	27.95	-13.87	-13.6639

The first test is applying the MLP on this series. Table 8 shows the results for the MLP training and test.

Table 8

MLP training and test results for d_1

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
98	1.22152	1.01913	0.081585	0.844187
99	1.22149	1.01911	0.08159	0.844198
100	1.22146	1.0191	0.081595	0.844208

We can see that the MLP behaves decent on this series, with a better than average prediction and a small total error. However a small overfitting occurs because the errors for the test slightly increase after more training.

Table 9, 10 and 11 show the results for the NAR-D architecture with an output regressor of 1, 2 and 3 respectively.

Table 9

NAR-D with b=1 training and test results for d_1

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
98	4.35932	0.926783	0.928081	0.869217
99	4.35939	0.92679	0.928063	0.869208
100	4.35945	0.926797	0.928045	0.8692

Table 10

NAR-D with b=2 training and test results for d_1

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
158	3.54816	0.896925	0.9006348	0.846096
159	3.5476	0.896853	0.9006226	0.84603
160	3.54733	0.896817	0.9006599	0.845997

Table 11

NAR-D with b=3 training and test results for d_1

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
158	3.02847	0.87418	0.8812488	0.822748
159	3.02847	0.874179	0.8812484	0.82274
160	3.02847	0.874179	0.8812479	0.822732

By using the output regressor, through the delayed targets, we can see that the performance of the system improves. Because the series has a recursive dependency of order 3, we can use a regressor of the same size for best results. In these scenarios, the network will use the extra information it needs in order to provide the best result. This kind of adaptive memory can be used for unknown series, but the user has to experiment in order to find the best result. In our case, because we know the dependency equation, we can apply and notice the difference in results.

The higher order tests require more epochs to converge because of lower learning rate.

To get a better idea about the quality of the prediction given by our predictor system, we can compare it with the random walk. Table 12 shows the test results for the random walk.

Table 12

Random Walk for d_1

f_1 RW training	f_1 RW test	f_4RW training	f_4RW test
7.34196	1.50122	1.20275	1.1055

We can see that the random walk, which is by definition a test that considers the previous value of the series, has less performance compared to the tested architectures above. The NAR-D prediction is much better, below average. We can also see that in the training the predictor estimates better, and in the test more previous values fed into the architecture improves the performance even

more.

Another type of architecture that we can test on this series is the NAR-Y. This type will only learn from it's own experience, using BPTT, without exogenous variables or delayed targets. The BPTT requires a smaller learning rate, so more epochs were necessary for convergence. Table 13 shows the results of NAR-Y with a feedback regressor of 1. Learning rate was 0.05.

Table 13

NAR-Y with b=1 training and test results for d_1

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
398	3.93048	0.880019	0.949667	0.879267
399	3.93048	0.880019	0.949667	0.879267
400	3.93048	0.880019	0.949667	0.879267

By changing the regressor size to 3, hidden units to 4 and learning rate to 0.04, we can see that the results for a *b = 3* NAR-Y do not have a significant change. Table 14 shows the results.

Table 14

NAR-Y with b=3 training and test results for d_1

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
224	3.93888	0.880958	0.903192	0.857482
225	3.93838	0.880902	0.903171	0.857472
226	3.93792	0.880851	0.903152	0.857464

We can see that even though the training errors are around the same, the test errors are smaller. The extra variables added by the two previous targets improve the performance.

Next test begins from the NAR-Y but also adds the delayed targets. This architecture is called NAR-DY and has no exogenous variables as well. The regressor order for feedbacks and delayed targets it's the same, *b*. In table 15 we can see the results with b = 1, 4 hidden units and a learning coefficient of 0.03.

Table 15

NAR-DY with b=1 training and test results for d_1

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
250	3.83782	0.870839	0.8397379	0.79954
251	3.83443	0.870455	0.8397377	0.799538
252	3.83223	0.870335	0.8397375	0.799536

We can see that adding the delayed targets improves the performance. The total errors are lower, and also the NRMSE. An interesting fact is that the errors are about the same for the training set, but better for the test set. Table 16 shows the results for the same architecture but with an output regressor of order 3.

Table 16

NAR-DY with b=3 training and test results for d_1

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
250	3.49907	0.855744	0.842327	0.809832
251	3.49907	0.855744	0.842327	0.809833
252	3.49907	0.855744	0.842327	0.809833

The results with a regressor of 3 do not improve. The squared error decreases a little for the training set, but for the test set it's virtually the same.

If we add the exogenous variables for the current time index, with the feedback, we obtain the NARX-Y architecture. The best results are obtained for a NARX-Y with 4 hidden units, a learning rate of 0.1 and a feedback regressor of 1. Table 17 shows the results for both training and test sets.

Table 17

NARX-Y with b=1 training and test results for d_1

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
250	4.65317	0.895499	0.716798	0.756675
251	4.65314	0.895497	0.716777	0.756664
252	4.65311	0.895494	0.716756	0.756653

We can see that by adding the exogenous variables, the performance increased. Once again, the exogenous variables show to be very important in obtaining a good prediction.

The same kind of experiment can be done with the other type of output regressor, for the delayed targets. This architecture is NARX-D. The results for the system with output order of 1, 3 hidden units and a learning rate of 0.2 are shown in table 18.

Table 18

NARX-D with b=1 training and test results for d_1

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
250	4.55408	0.88726	0.6661296	0.721043
251	4.55171	0.88513	0.6661416	0.721312
252	4.54815	0.88376	0.6661538	0.721586

The results for the architecture with delayed targets show to be much better than the ones with feedback. Table 19 shows the result with an output regressor of order 3.

We can see that the performance improves even more, which shows that previous targets assist in network training.

By comparing the two types of architecture, one can say that for this test function, the delayed targets are better than the feedback trained with BPTT, and better than the random walk. Also, the exogenous variables play a very important role in achieveing the results.

Table 19

NARX-D with b=3 training and test results for d_1

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
198	5.01653	0.981324	0.575028	0.662002
199	5.01699	0.98142	0.5750915	0.66225
200	5.01747	0.981629	0.575223	0.662761

There is no point in making any tests with TDNN-X because the function does not have recurrence in exogenous variables, or trying to use an input regressor.

A final interesting test is with the full NARX-DY architecture. The best results are obtained using a NARX-DY network with an output regressor of 3, with learning rate 0.06 and 11 hidden units. Table 20 shows the results.

Table 20

NARX-DY with b=3 training and test results for d_1

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
298	4.289213	0.914314	1.02334	0.914428
299	4.289212	0.914313	1.02334	0.914426
300	4.289212	0.914313	1.02333	0.914424

The prediction is not bad, is better than average for both training and test sets. The errors are also similar to the other architectures for the training set, but the prediction is worse than for NARX-D.

To summarize the results for the recurrent function d1, we obtained best results using a NARX-D architecture, with an output regressor of 3. This was expected as the function is depending on three previous values of the series, and has 3 exogenous variables. All this information fed into the system led to a better prediction than a normal MLP, which does not take into account order information. BPTT trained networks and combined architectures show good results, but not as good as the NARX-D.

5.3. Exogenous recurrent functions

In order to test TDNN-X performance compared to the other solutions, one needs a function that takes into account also previous values of the exogenous series, except the previous values of the series (targets). The function d_2 is an example of such a function:

$$d_2(t) = \sin(x(t) - y(t-2)) \cdot \log(y(t-1)+1) + \log(|d_2(t-1)|+1) - \sin(x(t) - y(t-1)) \cdot y(t); \quad (22)$$

From the formula of d_2 we can see that the series recurrence is 1 with respect to the targets *d*, and 2 with respect to the exogenous series *y*. There is no recurrence for the exogenous variable *x*.

The series average is E = 2.13767, variance = 112.326.

We can see in table 21 a sample of the series values, and in table 22 the random walk for the

training sequence (90%) and test sequence (10%).

Table 21

Recurrent exogenous function d_2

Index	Exogenous value (X)	Exogenous value (Y)	Series value (D)
1	0.000	3	0
2	0.002	3.05	0.439227
3	0.004	3.1	0.457068
...	...	...	...
52	0.102	5.5	-1.81446
...	...	...	...
447	0.890	25.25	-13.3032
...	...	...	...
500	0.998	27.95	27.4278

Table 22

Random Walk for d_2

f_1 RW training	f_1 RW test	f_4RW training	f_4RW test
0.0107479	0.00445453	0.0363901	0.0666476

Table 23

MLP training and test results for d_2

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
98	0.136959	0.147439	3.45377	2.23898
99	0.136923	0.14742	3.45494	2.23936
100	0.13689	0.147402	3.45612	2.23974

We can see that the MLP does a worse than average prediction, with a small overfitting because the errors for the test increase when the training errors decrease.

Now we can try the TDNN-X architecture, by adding the input regressor of size 2, in order to feed to the network the previous values of the exogenous variables. The results of the test with 3 hidden units can be seen in table 24.

Table 24

TDNN-X training and test results for d_2

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
98	0.564564	0.299295	2.44211	1.88272
99	0.565909	0.299652	2.44195	1.88266
100	0.567212	0.299996	2.44177	1.88259

The TDNN-X performes much better than the simple MLP. The exogenous variables improve significantly the results. We can also see that the NRMSE for this architecture is better than the random walk.

A surprise happens with the NAR-D architecture. A simple test of 3 hidden units, with a target regressor of 2 obtains a very good performance, much better than the average and closer to the random walk. Table 25 shows the results.

Table 25

NAR-D training and test results for d_2

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
298	0.0146229	0.0481764	0.118668	0.415022
299	0.0146025	0.0481429	0.118491	0.414711
300	0.0145827	0.0481102	0.118317	0.414408

By looking and the d_2 formula, we can see that the second term depends exclusively on previous series value. The results from NAR-D test indicate that this term is very important for the series, and the performance improves very much if we take this into consideration.

By combining the two types of architecture we will obtain NARX-D. A test with this architecture with learning coefficient 0.1, an output regressor of order 2 and an input regressor of order 2 is presented in table 26.

Table 26

NARX-D training and test results for d_2

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
298	0.0488415	0.0880436	1.97064	1.69125
299	0.0487325	0.0879454	1.96711	1.68973
300	0.0486239	0.0878473	1.96359	1.68822

The results show better performance than the TDNN-X but worse than the NAR-D. Adding the exogenous variables to the NAR-D did not improve it, but adding the targets to the TDNN-X improved. This indicates that the most important part in this function's prediction are the delayed targets.

Table 27

NAR-Y training and test results for d_2

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
25	4.62097	0.856416	0.829027	1.09695
26	4.37506	0.833317	0.859268	1.11678
27	4.17633	0.81417	0.888199	1.13543

Table 27 shows the results for the NAR-Y architecture with an output feedback regressor of 1. The results after 25 epochs show that the learning process is good, better than the MLP or random walk but worse than average prediction or the NAR-D. This indicates that the targets play a more important role than the feedback when predicting this series.

Table 28

NARX-Y training and test results for d_2

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
152	3.41212	0.735919	1.82292	1.62663
153	3.4116	0.735862	1.82222	1.62631
154	3.4111	0.735809	1.82155	1.62601

Table 28 shows the results by adding the exogenous variables to NAR-Y. The tested NARX-Y architecture performes better, showing that the exogenous variables improved the feedback. However the results are not as good as the NAR-D.

Table 29

NAR-DY training and test results for d_2

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
152	3.52675	0.915799	1.32235	1.13995
153	3.52674	0.915797	1.32235	1.13995
154	3.52672	0.915796	1.32235	1.13995

A hybrid combination between feedback and targets is the NAR-DY. By combining the targets and the feedbacks with a regressor of 2 we obtained the results shown in table 29. The overall performance is better than the NAR-Y with feedbacks alone, but it does not reach the level of the NAR-D.

Final test is the full NARX architecture, the NARX-DY. The results are shown in table 30.

Table 30

NARX-DY training and test results for d_2

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
58	39.819	2.51399	0.965195	1.18362
59	38.9743	2.4871	0.928564	1.16094
60	38.181	2.46173	0.899563	1.14267

The full NARX with an input regressor of 2 and an output regressor of 1, with learning rate 0.02 and 3 hidden units does not perform better than the more simple architectures. We can see that after 60 epochs the NARX-DY reaches the best performance which is closest to average but far away from the prediction given by NAR-D.

In the case of the function d_2, the targets play the most important role, while the feedback and exogenous variables degrade at some point the performance.

The synthetic tests of recurrency for both d_1 and d_2 show that the NARX architecture has the potential to obtain a good prediction but the user has to test all the different types with a training and test set and then decide which one is the best suited for the specific problem at hand.

5.4. Real data tests – Indian stock market

The first real data set that we are going to test is the Indian stock market for the index BSE30 for 1246 consecutive days. For each day (session), we have 4 values: the opening value of the index, the maximum value, the minimum value and the closing value. The fourth value is the target value that we want to predict, considering the other 3 as exogenous: the fourth value depends more or less to the opening , maximum and minimum values of the day.

The same rule for training and test applies: 90 % of the 1246 values will be used for training and the remaining 10 % for testing the performance of the predictor. The series average is E=11063.1, and variance Var = 1.74523e+07. Table 31 shows a sample of the data.

Table 31

BSE30 sample data

Index	Exogenous value (1)	Exogenous value (2)	Exogenous value (3)	Series value (D)
1	5868.39	5903.13	5843.77	5862.82
2	5878.17	5881.48	5852.13	5861.63
3	5875.53	5895.17	5782.37	5800.54
...	...	...	...	...
52	4717.83	4775.66	4680.42	4756.39
...	...	...	...	...
447	9293.8	9302.78	9158.44	9237.53
...	...	...	...	...
1246	11358.1	11367.2	10900.5	10947.4

Table 32

Random Walk for BSE30

f_1 RW training	f_1 RW test	f_4RW training	f_4RW test
1.46953e-07	3.53354e-08	0.037638	0.0917239

Table 32 presents the random walk values for the BSE30 index.

Different types of tests on this series have resulted in not so good results because the BSE30 is a very hard to predict index. For example, the NAR-Y architecture results are shown in Table 33. The feedback regressor was 2, the number of hidden units was 3 and the learning rate 0.1.

We can see that the training itself does not manage to get a very good error and NRMSE. The results for the test are worse. Because we don't know any equation or have any knowledge about the recurrence behind the numbers, it is very hard to pick the right architecture. All has to be tested by guess and see what kind of regressors or parameter values perform best.

Table 33

NAR-Y training and test results for BSE30

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
58	0.0134866	11.4022	0.446738	326.14
59	0.0115612	10.5569	0.446678	326.118
60	0.00991248	9.77526	0.446635	326.092

By adding the exogenous variables to the system, we obtain the NARX-Y. The results are better then the feedback only, but still unsatisfactory. The input regressor is 2, the output regressor is 1 and there are 3 hidden units. Table 34 shows the results.

Table 34

NARX-Y training and test results for BSE30

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
198	0.000118648	1.06947	0.225625	231.778
199	0.000118648	1.06947	0.225625	231.778
200	0.000118648	1.06947	0.225625	231.778

The training errors are much better and the prediction quality indicated by the NRMSE is almost as good as average. However the test results are far from that performance.

The best result from the performed tests is achieved by the NARX-D, with an input regressor of 2 and an output regressor of 1. Table 35 shows these results.

Table 35

NARX-D training and test results for BSE30

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
98	1.1263e-06	0.104466	6.2282e-06	1.21775
99	1.1263e-06	0.104466	6.2282e-06	1.21775
100	1.1263e-06	0.104466	6.2282e-06	1.21775

This time the errors in training set are very small, near to the random walk test. The test set performance is also much better, getting close to the average prediction.

Overall performance for the network is not very good, but the network learns and improves better than simple machine learning techniques, with possibility of future enhancement by performing more tests.

5.5. Real data tests – Inflation

Another series that the architecture is being tested upon is the US inflation in the period 1999 – 2012. The inflation is highly dependant on the unemployment rate and the exchange rate between US dollar and euro. For this reason, the two variables are exogenous, if we want to predict the inflation. The data consists of 161 monthly values, of which 90 % will be used for training and 10 % for test. All data is from Eurostat[2]. Table 36 shows a sample of the data.

Table 36

Inflation sample data

Index	Exchange rate US dollar to Euro (1)	Unemployment rate in percent (2)	Inflation rate in percent for previous 12 months (D)
1	1.1608	4.8	1.2
2	1.1208	4.7	1.3
3	1.0883	4.4	1.3
...	...	...	...
22	0.8721	3.8	3.4
...	...	...	...
83	1.2015	4.6	5.7
...	...	...	...
161	1.2789	7.5	2.3

Table 37

Random Walk for inflation

f_1 RW training	f_1 RW test	f_4RW training	f_4RW test
8.42935	0.302525	0.329362	0.481856

After testing the series in the simulator, a few interesting results are worth to mention. The NAR-D performes very good, with a regressor of 2 , with 3 hidden units and a 0.2 learning coefficient. The results are shown in Table 38.

Table 38

NAR-D training and test results for inflation

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
141	8.03675	0.3216	0.609282	0.683826
142	8.03606	0.321586	0.609329	0.683852
143	8.03537	0.321572	0.609378	0.68388

2 http://epp.eurostat.ec.europa.eu/portal/page/portal/eurostat/home/

We can see that the NAR-D does a good prediction, better than average but not as good as the random walk. By adding the exogenous variables, the architecture NARX-D improves significantly. Table 39 shows the results of the test with an output regressor of 2, an input regressor of 1.

Table 39

NARX-D with a = 1 training and test results for inflation

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
141	8.69904	0.336162	0.247738	0.436046
142	8.69774	0.336137	0.24857	0.436778
143	8.69642	0.336111	0.249447	0.437548

Even though a small overfitting occurs, the results are much better, and better than the random walk test. In order to find out if the series has a more deep recurrence, another test is made with an input regressor of order 3. Table 40 shows the results.

Table 40

NARX-D with a = 3 training and test results for inflation

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
141	7.66278	0.318153	0.277358	0.461378
142	7.65916	0.318078	0.276452	0.460624
143	7.65543	0.318	0.275728	0.46002

The results show no significant change for the performance. This indicates that the extra exogenous variables that are being fed into the system (the older than 2 time steps) are not actually useful for the current time step and they have no significant role in the inflation. Only the first two older values improve the prediction.

In order to see if the exogenous variables play a more important role in the prediction, we can try the TDNN-X achitecture, with exogenous variables only. Table 41 shows the result of such architecture, with an input regressor of size 2, applied over our set of data.

Table 41

TDNN-X training and test results for inflation

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
141	18.6962	0.494875	6.9059	2.30222
142	18.6989	0.494911	6.9059	2.30222
143	18.7015	0.494946	6.90589	2.30222

From what we can see, the exogenous variables are not very important, but they play a good role in improving the NAR-D. Thus, the previous values of the inflation have a more significant impact rather then the previous values of unemployment and exchange rate.

The prediction given by NARX-D is the best from the tested architectures and it provides a

good forecast for the problem.

5.6. Real data tests – Hidrology

An interesting application for the simulator is the following series: the level of the lake Huron, one of the 5 great lakes of North America. We will use another series, which is the measured annual precipitation rate in inches, from 1900 to 1986. The level of the lake water is measured as mean of the month July, annual, on the Harbor beach, in feet. All data is retrieved from Rob Hyndman website[3].

Intuitively, the rain should affect in a great measure the level of the lake. However this is the total annual precipitation level, so if we measure the lake level in july, some precipitations after this month will not affect current year, but maybe the next year measurement.

Table 42 shows a sample of the data, which consists of 87 values, of which 90 % are going to be used for training and 10 % for test.

Table 42

Huron lake series

Index	Precipitation level, inches(X)	Lake level, feet (D)
1	31.42	578.82
2	30.28	579.32
3	33.21	579.01
...	...	...
12	30.05	578.69
...	...	...
34	35.11	576.9
...	...	...
87	36.36	581.27

Table 43

Random Walk for lake Huron

f_1 RW training	f_1 RW test	f_4RW training	f_4RW test
23.2091	1.45008	0.505614	0.924954

The MLP results for the Huron series are shown in table 44. We can see that the prediction is fair, and around the performance of the series average. The random walk itself it's not very far away from the average. This suggest that the rainfall has some importance, but further tests will give us more insight.

3 http://robjhyndman.com/TSDL/

Table 44

MLP training and test results for Huron series

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
28	47.375	0.724246	1.79749	1.02981
29	47.389	0.724353	1.79275	1.02845
30	47.4002	0.724438	1.78909	1.0274

If we take into consideration the rainfall from the previous 2 years as well, by adding an input regressor of value 2, we obtain the TDNN-X, and the results after 200 epochs are presented in table 45.

Table 45

TDNN-X training and test results for Huron series

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
198	35.6102	0.625225	0.324416	0.437498
199	35.6084	0.625209	0.324717	0.4377
200	35.6065	0.625192	0.325016	0.437902

The results are very impressive, the precipitations in the previous years influence very much the level of the lake, we can see that the prediction is much better than the average and much better than the random walk.

If we take into consideration just past values of the lake level, we obtain the NAR-D architecture. The results for an output regressor of 2 are available in table 46.

Table 46

NAR-D training and test results for Huron series

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
98	20.1132	0.471902	1.63752	0.982921
99	20.1109	0.471875	1.63791	0.983036
100	20.1087	0.471849	1.6383	0.983154

Even though the errors and prediction quality for the training set is much better, the system performes worse on the test set. This indicates that the previous lake levels are important for the next year, but not as important as the rainfall. By combining the two types of regressors we obtain the NARX-D architecture. The results of a NARX-D test with an input regressor of 2 and output regressor of 2 are shown in table 47.

We can see that the performance is better than simple NAR-D, but not as good as the TDNN-X. We conclude by saying that the lake levels depend greatly on the rainfalls of the previous 2-3 years but also on the level of the lake in the previous years but not in such a great order. The prediction offered by the system is very good and better than the random walk tests or average.

Table 47

NARX-D training and test results for Huron series

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
22	13.6438	0.387004	0.833285	0.701167
23	13.5864	0.38619	0.830057	0.699807
24	13.5402	0.385533	0.828709	0.699239

5.7. Real data tests – Temperature

This test consists of two series, the outdoor temperature measured in Celsius degrees , and the indoor temperature of a house, also in Celsius degrees. The data is measured for 168 consecutive days. Source of the data is Rob Hyndman website[4]. In this scenario we can consider the outdoor temperature as an exogenous variable for the indoor temperature. The purpose is to check how many days in the past influence the current indoor temperature. If, for example, the outdoor temperature rise, we should see the effect later, because of walls being warmed up slowly. Viceversa, if the outside temperature would drop, we could see the inside temperature drop after a period of time. The target series average is E = 24.1975, and variance Var = 2.01865, while for the exogenous series, E = 17.1571, Var = 36.4426. Table 48 shows a sample of the data.

Table 48

Temperature series

Index	Outdoor temperature, Celsius (X)	Indoor temperature, Celsius (D)
1	15.5667	25.2778
2	14.8611	25.0889
3	14.2222	24.9444
...	...	...
12	23.2055	24.75
...	...	...
34	12.4556	22.8722
...	...	...
168	19.9167	26.7944

Table 49

Random Walk for temperature

f_1 RW training	f_1 RW test	f_4RW training	f_4RW test
7.92325	0.612006	0.251651	0.251221

4 http://robjhyndman.com/TSDL/

The MLP in this case will only take into consideration the outdoor temperature in the current day. The results are in table 50.

Table 50

MLP training and test results for temperature

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
98	56.5164	0.672101	8.44408	0.933156
99	56.5108	0.672068	8.43882	0.932866
100	56.5054	0.672035	8.43363	0.932578

We can see that the MLP prediction is better than average but worse than the random walk, which is just predicting that the same temperature for the previous day will happen.

Table 51

TDNN-X with a = 2 training and test results for temperature

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
14	33.4156	0.516428	4.30022	0.665922
15	32.1907	0.506874	4.25548	0.662449
16	30.9982	0.497397	4.24311	0.661485

By using the TDNN-X architecture with an input regressor of size 2, we obtain much better results, that get closer to the random walk performance. This indicates that indeed the 2 previous days outdoor temperatures influence highly the indoor temperature of the current day.

Table 52

TDNN-X with a = 7 training and test results for temperature

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
38	14.766	0.34154	4.67084	0.694026
39	14.7663	0.341544	4.66141	0.693325
40	14.7668	0.341549	4.65247	0.692659

We can see that if we increase the input regressor to 7, the TDNN-X performance does not improve. This suggest that the outdoor temperatures older than 3 days do not influence the current indoor temperature.

However, the previous indoor temperatures may have an effect as well. By adding previous target values we will test the NARX-D architecture. Results are shown in table 53.

We can see that the previous indoor temperatures have a very high influence of the current day. The test has been performed with an input regressor of 2 and an output regressor of 2. The values of the previous 2 days for both series have high influence, while the older values have little or no influence at all.

The results are much better than the average and better than the random walk prediction.

Table 53

NARX-D training and test results for temperature

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
148	4.43168	0.18807	0.310921	0.179062
149	4.43154	0.188067	0.310927	0.179064
150	4.43141	0.188064	0.310934	0.179066

If we try to use the full NARX architecture, NARX-DY, with input and output regressors of 2, trained by BPTT, we obtain good results, almost as good as random walk, but worse than the NARX-D. We can see the results in table 54.

Table 54

NARX-DY training and test results for temperature

Epoch no	f_1 SSE	f_4 NRMSE	f_1 SSE test	f_4 NRMSE test
998	5.28118	0.205453	2.76245	0.533734
999	5.28016	0.205433	2.76243	0.533733
1000	5.27914	0.205413	2.76242	0.533732

To conclude, for the temperature series, the simulator offers a very good prediction, given especially by the NARX-D architecture.

6. Conclusions and future work

The project provides a thorough research in the sequence task processing focusing on time series prediction using the NARX model. The different types of NARX architectures are being studied and tested, starting from the basic MLP and ending with the most complex NARX having feedback trained with BPTT , delayed target regressors and delayed exogenous input regressors.

Several types of tests of performance are being proposed, beginning with a basic root mean square but also having normalized errors and distribution fitness tests. This tests allows the user to have a simple measurement of how good is the prediction given by the system, and to tweak different parameters in order to improve his work.

The application is a NARX simulator that can generate artificial time series using a generator or predefined series, or load real data series from files. The simulator then allows the user to select the desired type of architecture and parameters, and train a model specific for the problem at hand, or test the simulator using the predefined ones.

Further, the user can see the tests results for each epoch both for training and test sets, and also for random walk tests. The user can also use a manual prediction module for tests.

The performance of the simulator is analyzed using basic test functions, that are most suited for the MLP, but also with recurrent functions that make use of the output regressors, for the recurrency of the targets, and recurrent functions with respect to the exogenous variables that use the input regressor. For each type of functions, the different type of architectures are tested with varying the most important parameters.

The simulator is also tested for various real data series, from finance, hidrology or temperature measurement. The performance and the tests indicate that the simulator performes well, with the necessary tweaking for the specific problem. Every problem needs to be tested in the different architectures offered, and a best fit needs to be found. Tests for the indian BSE30 have been made, for US inflation, precipitation and lake Huron levels, and for indoor/outdoor temperatures.

Overall the performance of the simulator is good and the results are better than an average prediction or a random walk.

The application provided is a useful tool for any specialist in the field who uses sequences and wants to predict further values. The applicability of the program includes finance, economy, agriculture, weather forecasting, chemistry of physics.

A possible approach for the future work would be to make the NARX network adaptive, by trying to add or remove the certain inputs that cause the errors to increase. Varying the parameters dynamically is also possible, so that the user does not have to manually adjust for finding the best model for the problem, but let the program test and learn by experimenting. A meta-neural network or some other type of learning mechanism can be used for this process.

Another possible future work is trying to avoid saturation and overfitting by using regularization, by using penalization terms[9].

The simulator itself can also be improved by adding more options, more activation functions, or more options to the logging or evaluation module, different functions for performance evaluation, etc.

B I B L I O G R A P H Y

[1] M. C. Mozer, “Neural net architectures for temporal sequence processing,” in *Santa Fe Workshop*, 2007, vol. 15.

[2] J. D. Hamilton, *Time series analysis*, vol. 2. Cambridge Univ Press, 1994, pp. 18-1 to 18-63.

[3] J. C. Principe, J. M. Kuo, and S. Celebi, “An analysis of the gamma memory in dynamic neural networks.,” *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 331-337, 1994.

[4] P. Werbos, “Backpropagation through time: What it does and How to do it,” *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550-1560, 1990.

[5] M. Bodén, “A guide to recurrent neural networks and backpropagation,” *The DALLAS project. Report from the NUTEK-supported project AIS-8, SICS. Holst: Application of data analysis with learning systems*, no. 2, pp. 1-10, 2001.

[6] T. Lin, B. G. Horne, P. Tino, and C. L. Giles, “Learning long-term dependencies in NARX recurrent neural networks.,” *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, vol. 7, no. 6, pp. 1329-38, Jan. 1996.

[7] H. Akaike, “A new look at statistical model identification,” *IEEE Transactions on Automatic Control*, vol. 19, no. 6, pp. 716-723, 1974.

[8] S. Ingrassia, “Equivalent number of degrees of freedom for neural networks,” *Advances in Data Analysis*, no. 1, 2007.

[9] K. Fukumizu, “Likelihood ratio of unidentifiable models and multilayer neural networks,” *The Annals of Statistics*, vol. 31, no. 3, pp. 833-851, 2003.

[10] A. N. Kolmogorov, “Sulla determinazione empirica di una legge di distribuzione,” *G Inst Ital Attuari*, vol. 4, pp. 83-91, 1933.

[11] T. Anderson and D. Darling, “Asymptotic theory of certain ‘ goodness of fit’ criteria based on stochastic processes,” *The Annals of Mathematical Statistics*, 1952.

[12] H. T. Siegelmann, B. G. Horne, and C. L. Giles, “Computational capabilities of recurrent NARX neural networks.,” *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*, vol. 27, no. 2, pp. 208-15, Jan. 1997.

[13] J. Menezesjr and G. Barreto, “Long-term time series prediction with the NARX network: An empirical evaluation,” *Neurocomputing*, vol. 71, no. 16-18, pp. 3335-3343, Oct. 2008.

[14] E. Diaconescu, “The use of NARX neural networks to predict chaotic time series,” *WSEAS Transactions on Computer Research*, vol. 3, no. 3, pp. 182–191, 2008.

Annex 1 – Simulator screenshots

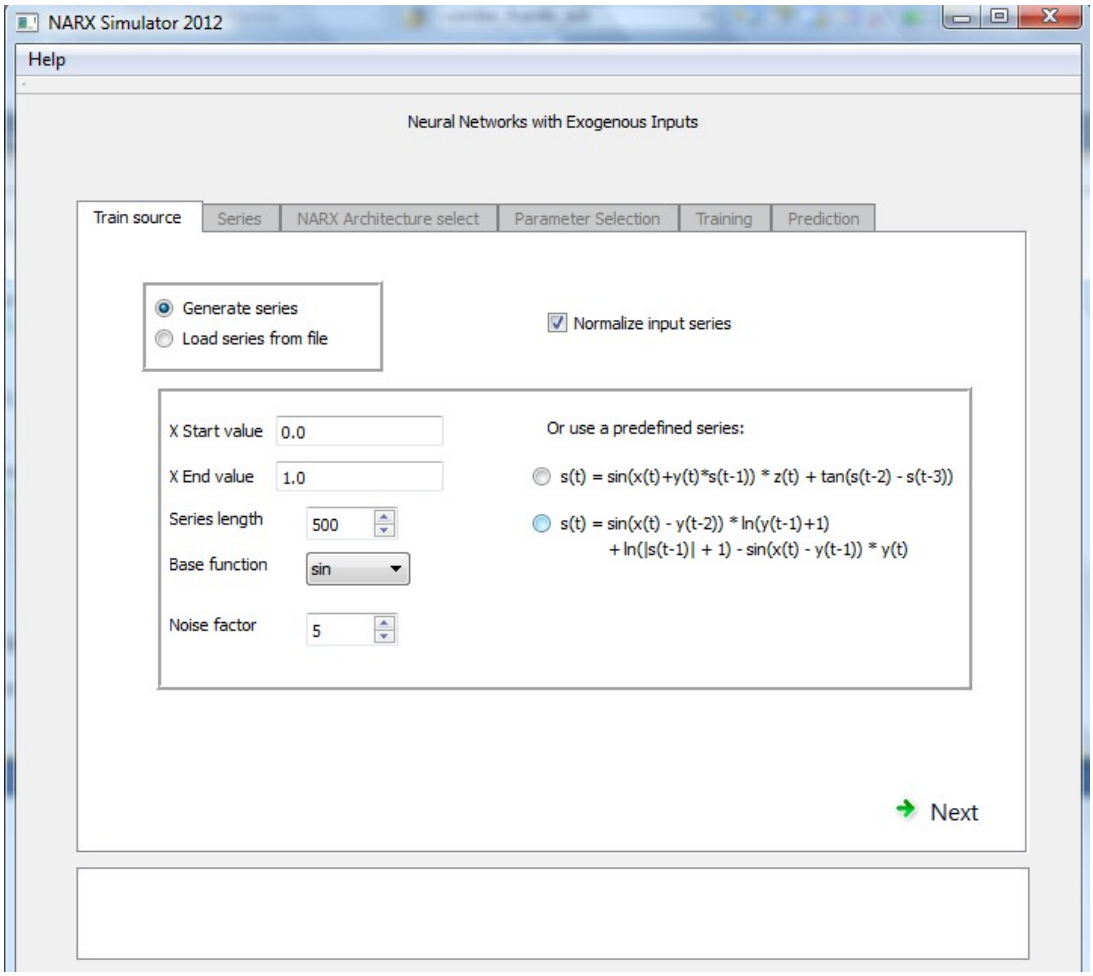

Fig. 10. First page of the simulator: train source

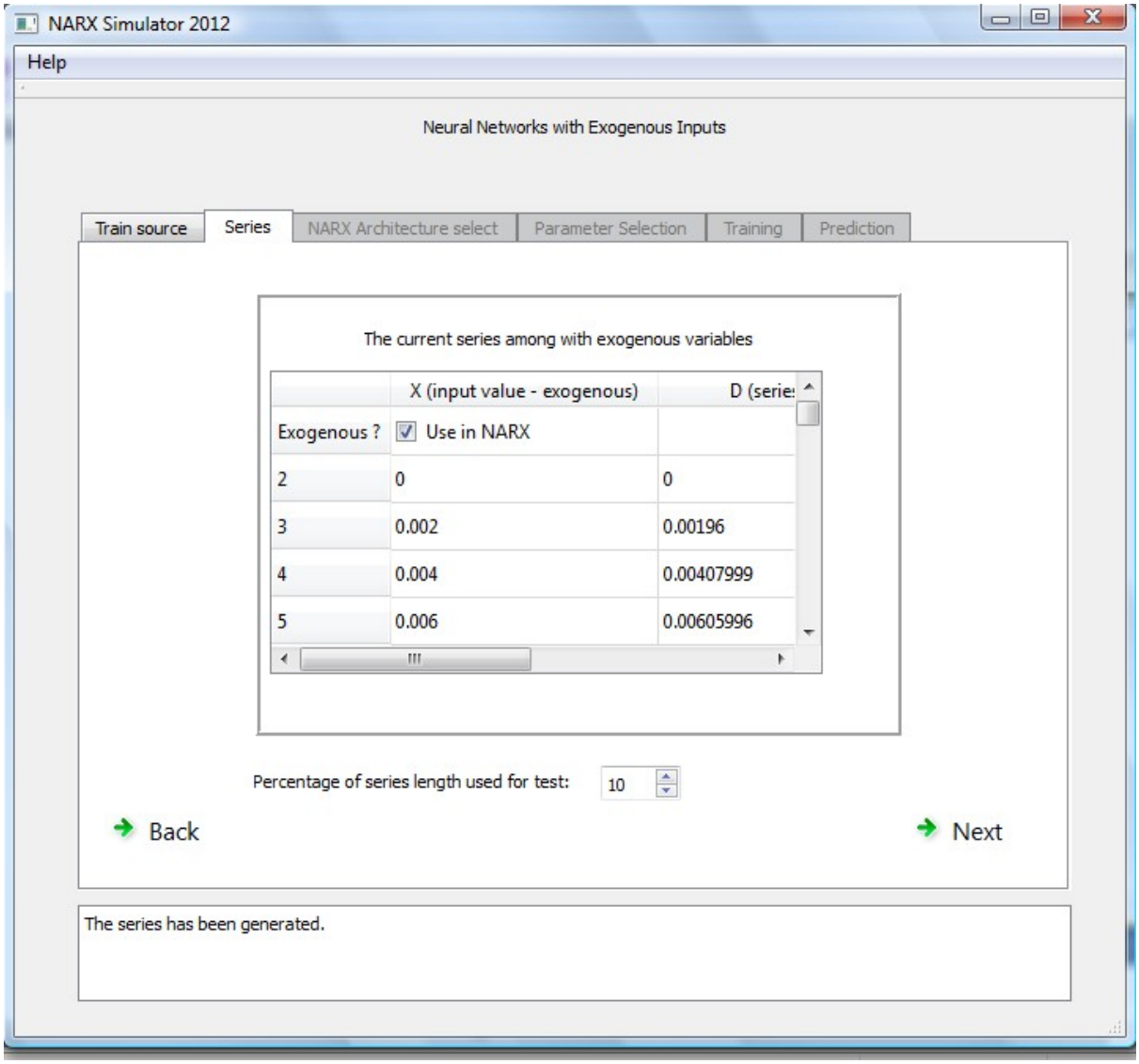

Fig. 11. The series screen

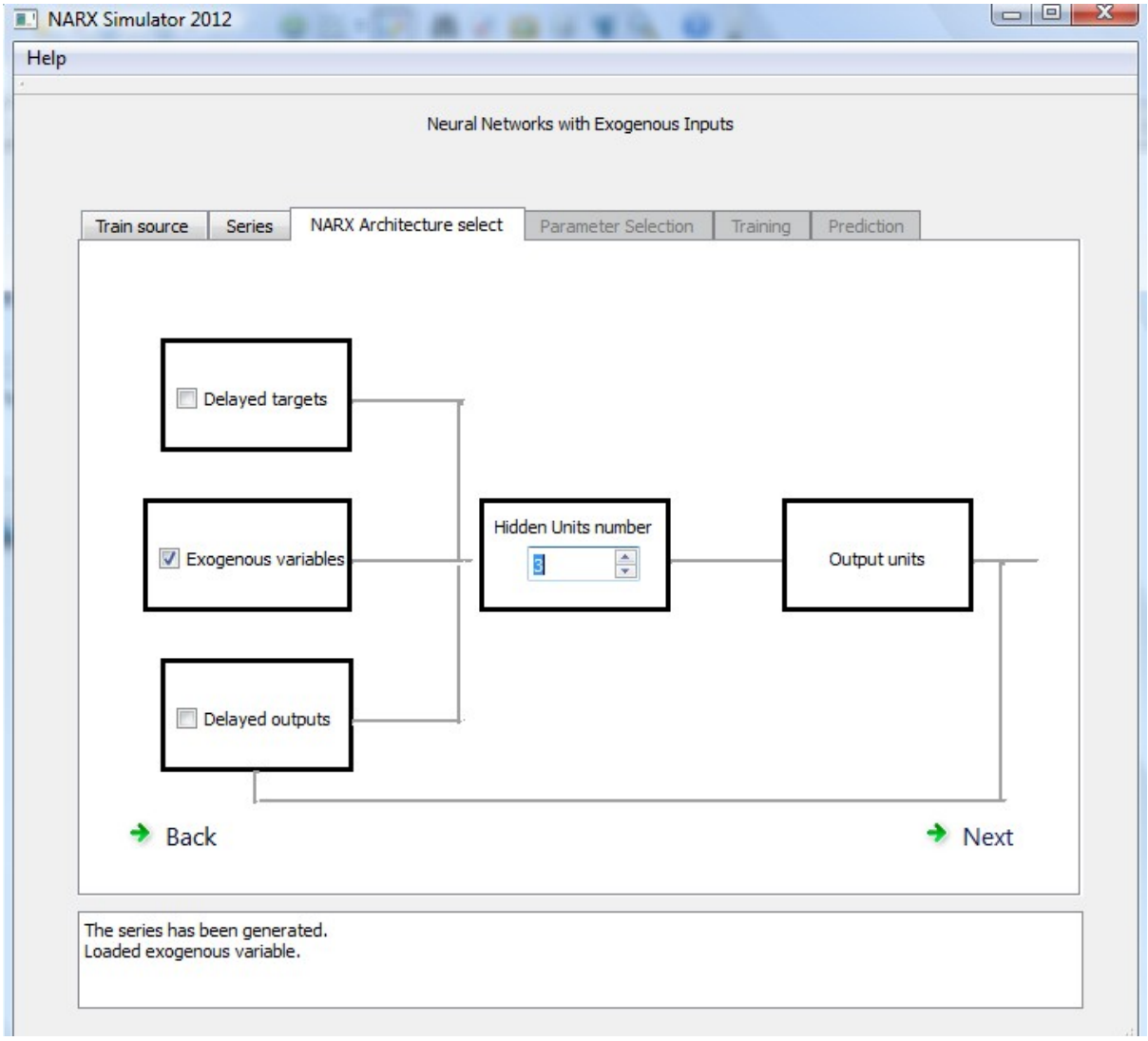

Fig. 12. NARX architecture select page

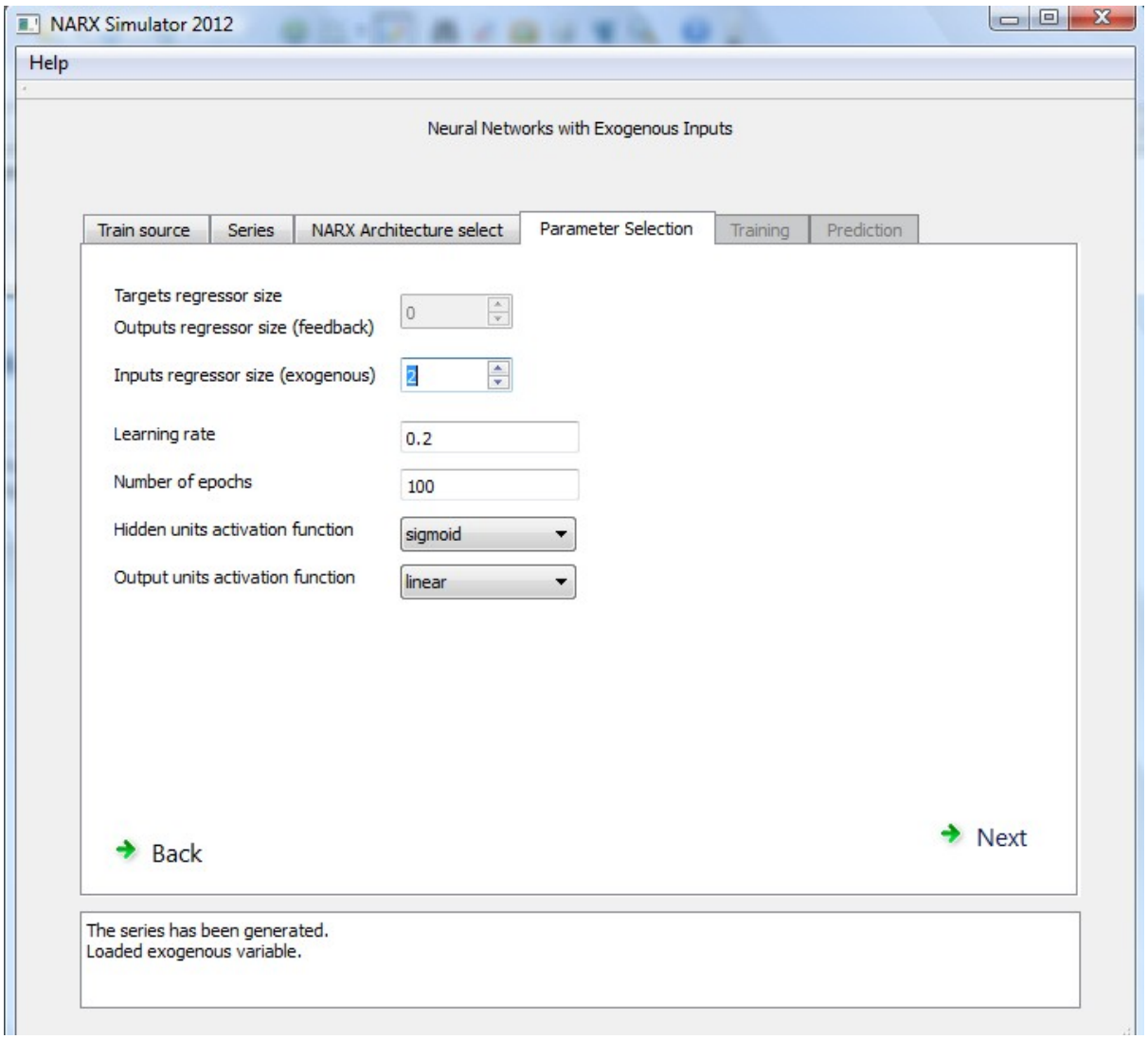

Fig. 13. NARX parameter customization page

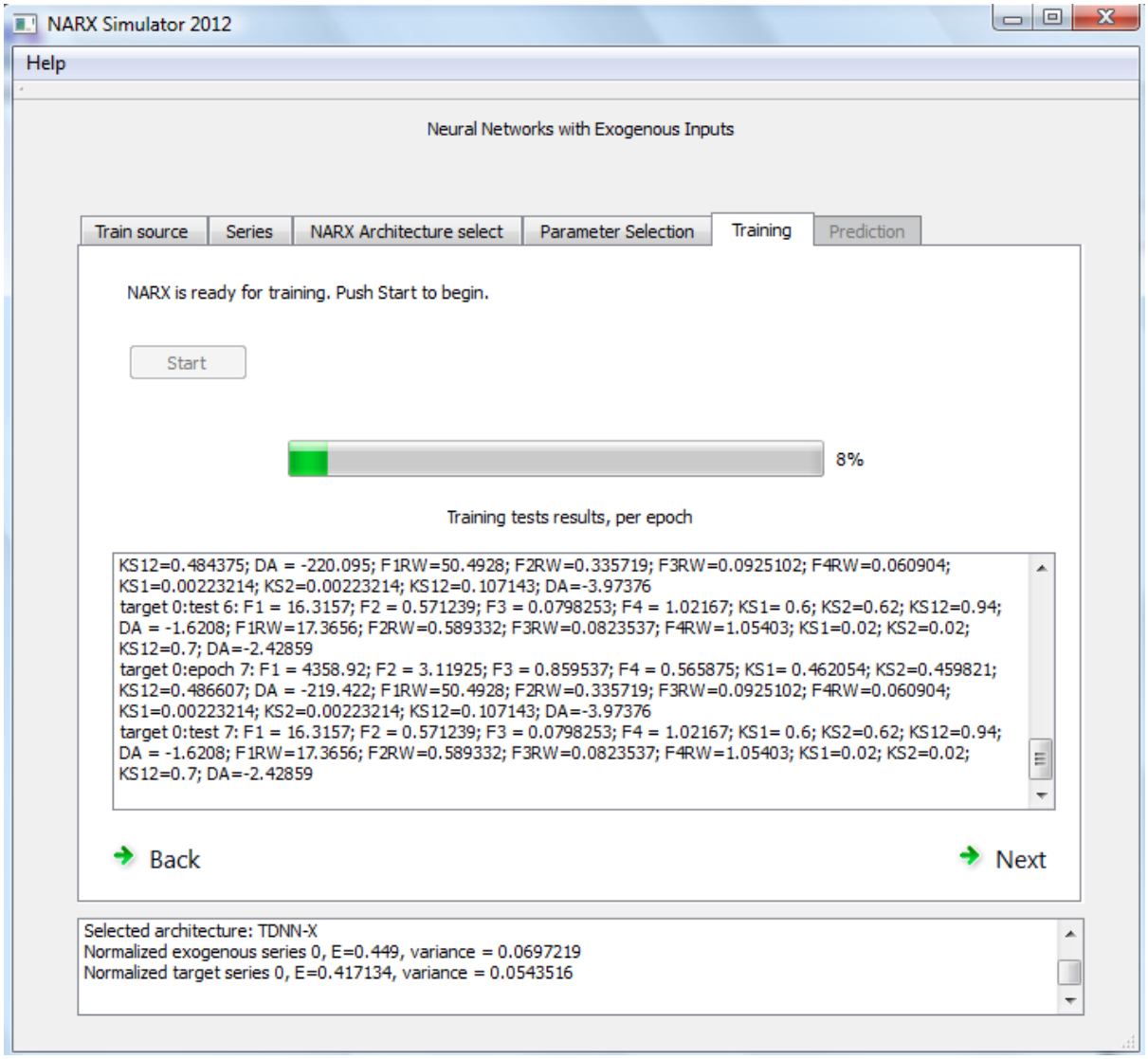

Fig. 14. Training phase

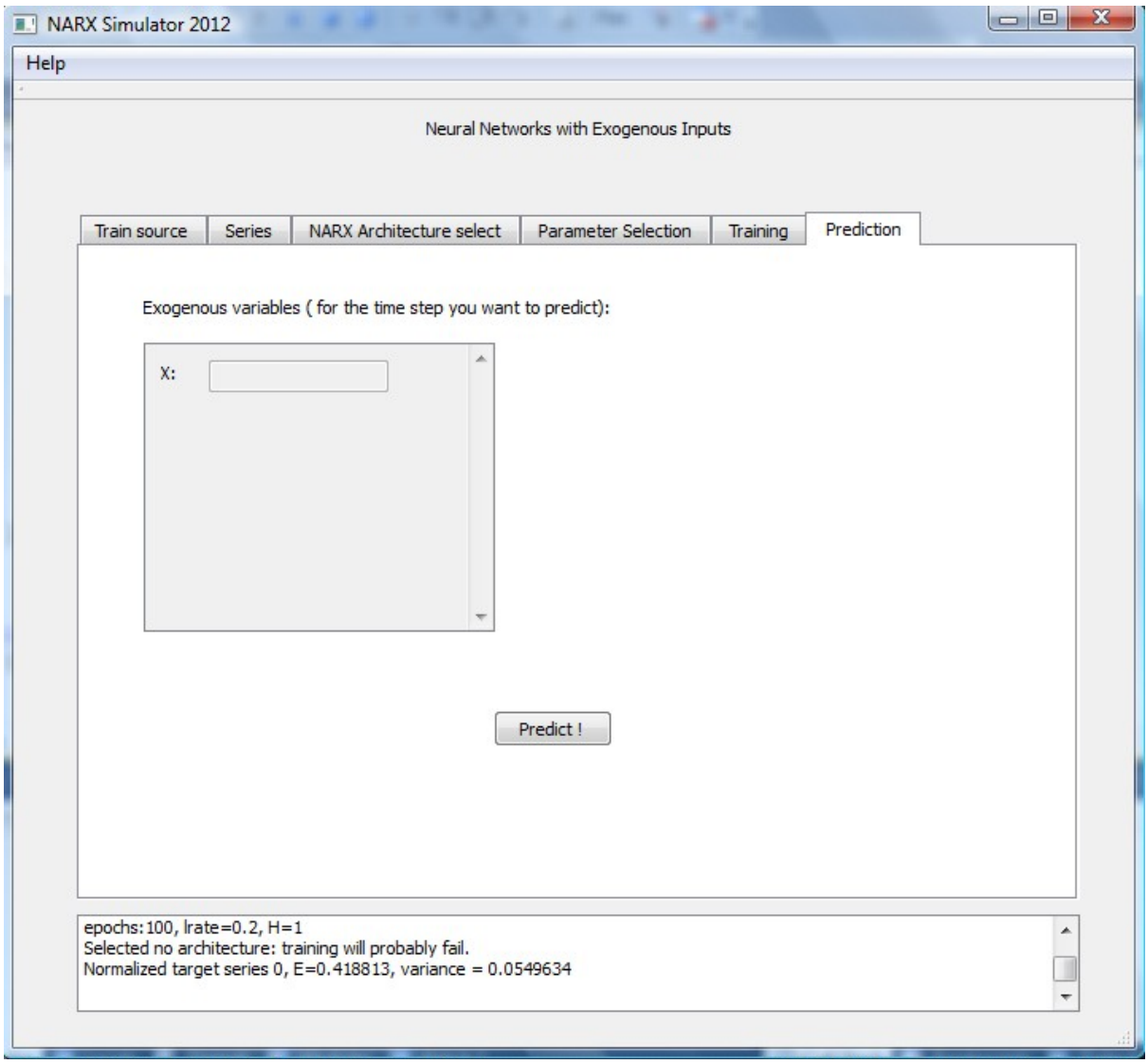

Fig. 15. Prediction phase