CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF MECHANICAL ENGINEERING

Department of Instrumentation and Control Engineering
Division of Automatic Control and Engineering Informatics

Bachelor's Thesis

Comparison of Neural Network Models for Approximation of Pneumatic Muscle Actuator

Author: Zhuldyz Assylova
Supervisor: doc. Ing. Ivo Bukovsky, Ph.D.

Academic Year: 2012/2013

Statement

I declare that I have worked out this thesis independently assuming that the results of the thesis can also be used at the discretion of the supervisor of the thesis as its co-author. I also agree with the potential publication of the results of the thesis or of its substantial part, provided I will be listed as the co-author.

Prague, ___________________

Signature__________________

Acknowledgement

The author and the thesis supervisor would like to thank their colleagues from *Technical University of Košice*, namely

doc. Ing. Ondrej Líška, doc. Ing. Ján Piteľ, Ph.D. and Ing. Alexander Hošovský, PhD

for the possibility and their cooperation on the research of pneumatic muscle actuator that has been originally supported by project:

VEGA 1/1162/11 Teoretické princípy, metódy a prostriedky diagnostiky a rehabilitácie mobility seniorov

and by the

Project of the Structural Funds of the EU, Operational Programme Research and Development, Measure 2.2 Transfer of knowledge and technology from research and development into practice. Title of the project: Research and development of nonconventional pneumatic artificial muscles-based actuators. ITMS code: 26220220103.

The author also would like to thank her supervisor Doc.Ing. Ivo Bukovsky, Ph.D. for his excellent supervision and valuable suggestions during the work on this Bachelor's thesis.

Abstract

This bachelor’s thesis is a case study of approximation of the pneumatic muscle actuator model (PMA) in antagonistic connection with pulse width modulation (PWM) control input by means of artificial neural networks. Three different neural network architectures and their static and dynamical version were derived and programmed (open-source, Python) and tested for their ability to handle nonlinearity of PMA with PWM.

Contents

Notation

k … discrete time index (with constant sampling)
u … control input
$\mathbf{x}$ … general input vector
y … controlled variable
y_n … neural output

1 Introduction

When it comes to robot human interaction main things people must think of safety, compatibility and ease of operation. Common pneumatic cylinders are usually exploited to actuate robots to mimic some basic human motions. But it has been realized that because of their size and large weight these actuators are not very suitable and safe for most of the applications. That is why it’s more essential to use Pneumatic Muscle Actuators (PMA) that are very light weight, small-sized and compliant which make them suitable for applications involving robot human interaction. However, there are some certain shortcomings that make them not so widely used; their highly nonlinear and time dependent behavior can cause some crucial problems.

This thesis is a case study of approximation of PMA model [8][9] using various neural networks when inflation and deflation is controlled by two valves with single control input via pulse width modulation [24].

In the first section the design, construction and experimental testing of the pneumatic artificial muscles developed in different arrangements is reviewed. I also described the various methods of modeling PMA with a strong emphasis on the artificial intelligence approach.

The main objective of the further work is to design the neural networks, capable of effectively approximating the given model of pneumatic muscle actuator; three different neural network architectures that would be trained and tested are proposed. In order to tune and optimize the networks parameters the suitable learning algorithms and an appropriate initial setup is chosen.

Furthermore, I experimentally compared the designed neural networks in terms of efficiency and accuracy by calculating the values of Sum of Squared Errors and the Coefficient of Determination and most importantly in respect to their ability to approximate the specific nonlinearity of PWM actuated PMA [24].

The final goal is the summarization of all obtained results and their discussion, and to conclude whether the chosen methods of approximation are suitable and can be applicable in the field of PMA control.

2 State of the Art

2.1 Review of Pneumatic Muscle Actuators

As one can see from the recently published papers Pneumatic Muscle Actuators (PMA) have been widely used in the various fields such as industrial applications and robotics and apparently become quite popular over commonly used pneumatic cylinders and linear actuators. Apart from advantages mentioned above, the main reason of their increasing use is that these actuators are simple and easy to construct and predict. Typically, they are constructed from a thin walled rubber (latex or non-vulcanized rubber) cell and a braided sheath [1]**.** The left side of the rubber tube is shut while an air hose is implemented into the right end to inflate or deflate. Once the inner part of the rubber tube is inflated, it tends to expand and the diameter of the rubber-sheath, called usually the "muscle", effortlessly increases. If one attaches a mechanical load to this end of the muscle and there will be an external work done at a higher rate on the load, he can end up achieving very high power rate. Since PMA are light in weight and most likely flexible, they can provide safe and soft interaction due to high performance and are more suitable for robot-human interactions applications. Clearly, they also have rather high power to weight ratio and big volume to weight ratio thus are suitable especially for wearable robotic applications [1]. However, despite the mentioned advantages, PMA have not been extensively used in the past due to its inherent drawbacks such as complexity of controlling and unpredictability of the system. Still, recent results show that there are certain ways of solving this task, for instance, neural network technique seems to be very effective to identify a broad category of complex problems concerning PMA. Summarizing, pneumatic muscle actuator is the general type of the family of inflates-deflates tube-like actuators that are designated by a decrease in actuator length when the pressure is applied and they have more significant advantages compared to other types of actuators.

2.1.1 Modeling PMA with a Spring

In paper [2] authors are dealing with a driving system that is supposed to be large when it comes to the large compressors which are usually exploited along with pneumatic actuators. They proposed the low-pressure, low-volume pneumatic actuator for manipulating a robot hand that can be used safely by people and a five-fingered robot hand where they would install those actuators.

The driving force of the pneumatic artificial muscles differs when air becomes input from when air is output. It occurs due to the hysteresis characteristics of the pneumatic actuators [2]. To analyze that, authors in [2] built a model by using a spring element (see Fig.1), which signifies the driving force of the pneumatic pressure, and a damper element, which denotes the hysteresis part. They constructed a 1-link arm with one degree of freedom using pneumatic actuators; and also implemented a Proportional–Integral–Derivative (PID) controller system in order to check the efficacy of the model in an experiment (see Fig.2). Each parameter of PID controller was defined so that integral squared error value is minimized and the overshoot becomes 5% or less of the actual value.

The gains of the system were determined with a simulation that uses the mentioned above 1-link arm model, and after that using the obtained data the step response of the joint angle was examined.

Authors also derived the characteristics of the relationship between the driving force and displacement obtained in the experiment at a pressure 100 kPa and velocity of 10×10-3 m/s. They assumed that the characteristics of the pneumatic actuator are easier to be analyzed this way. The coefficient of determination when the multiple regression analysis was performed was equal to 0.95, which is a very high value showing the success of the experiment [2].

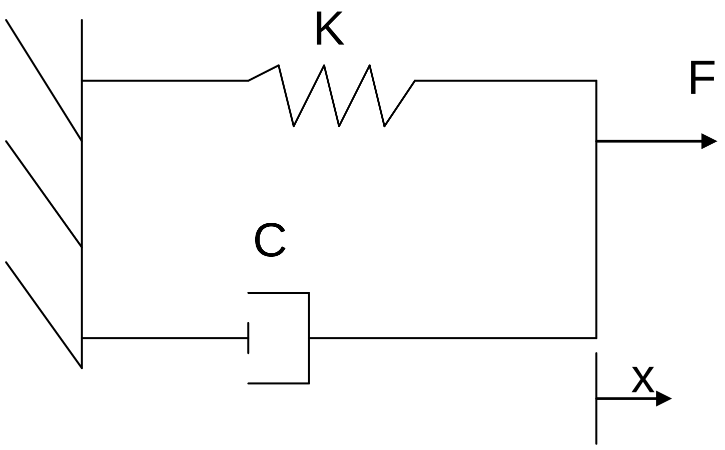

Fig.1 Spring and damper element model of pneumatic actuator, as shown in the paper [2]

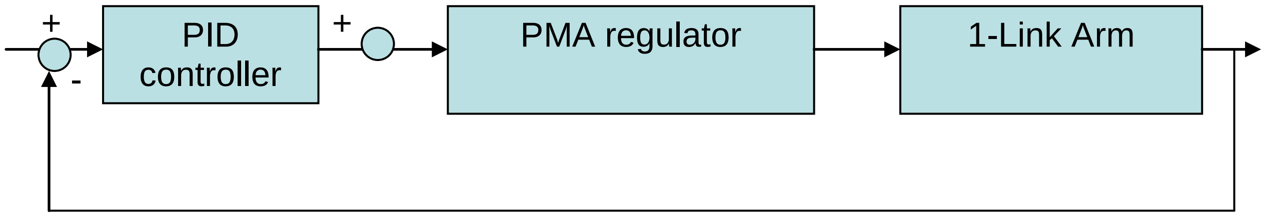

Fig.2 PID controller system (Experiment), as illustrated in [2]

2.1.2 Arrangement in Antagonistic Connection

Pneumatic muscles actuators can be also connected in the antagonistic manner. The pneumatic muscles act one against another and the final position of the actuator is given by steady state of all the forces according to different pressures in muscles. In papers [3][4] authors designed a model of the system consisting of ON/OFF solenoid valves and pneumatic muscle actuators in antagonistic connection. The designed model of the antagonistic actuator responds with different dynamics of the pneumatic artificial muscle by inflation and deflation of compressed air into or from muscle.

Work of the designed PMA-based antagonistic actuators was realized by applying pressure in one of the artificial muscles and at the same time reducing pressure in the other (antagonistic) artificial one. Since both artificial muscles are active, they require synchronous pressure adjustment in both muscles. It’s a very complex procedure because the equilibrium condition between air pressure (/volume) increment in one artificial muscle and air pressure (/volume) decrement in the other artificial muscle has to be performed. Otherwise they could obtain the uneven motion of the actuator arm.

Generally speaking, authors designed the antagonist actuator on the basis of theoretical analysis of the actuator function, mathematical description of the main actuator components and experimental measurements using the block diagram. The main drawback of PMA based antagonistic actuator is its non-linearity of the system, which causes the non-linearity of the end position of air filling pressure vessel in the muscles.

Paper [4] also specifically suggests a dynamic model of the system consisting of four ON/OFF solenoid valves and two pneumatic artificial muscles in antagonistic connection. This model deals with different dynamic characteristics of the pneumatic artificial muscle actuated by inflation and deflation of compressed air into or from muscle.

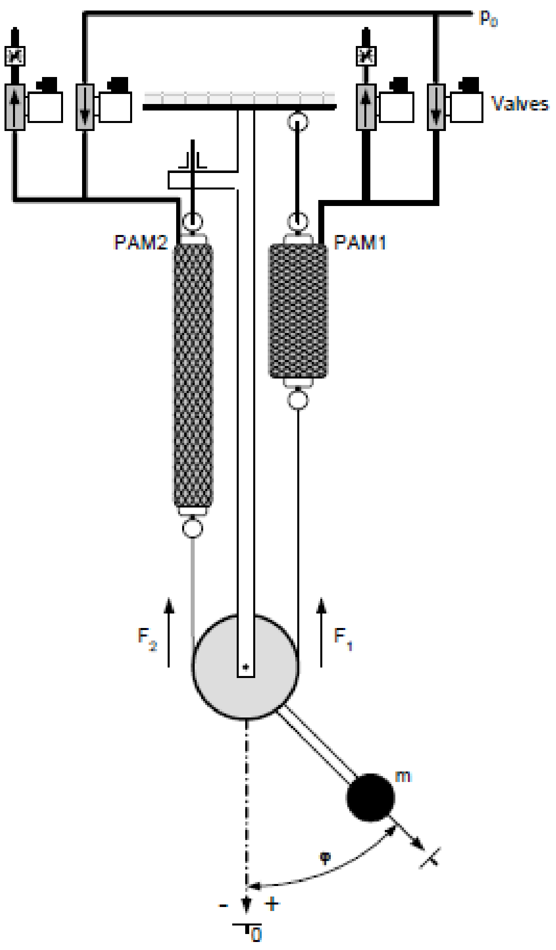

Fig.3 The experimental setup for testing the antagonistic actuator, adopted from [3]

The model in [4] was created in Matlab's Simulink software, and the main part of simulation model of pneumatic artificial muscle based actuator was subsystem of muscle nonlinearity and its dependence between pressure in muscle and air flow rate into or from a muscle. Another part of model is a subsystem of solenoid valve, where dependence between air flow rate and pressure difference in front of the valve and behind the valve was also used. Eventually they used a subsystem of actuator nonlinearity based on approximation of the measured static characteristics of the realized pneumatic muscle actuator for creating the model. Obtained model was simulated with various time dependent control signals of the inlet (outlet) solenoid valves and with different muscles parameters.

2.2 Computational Intelligence Methods for PMA

This subsection reviews mainly neural network approaches for PME; however, some other approaches as fuzzy systems or hybrid ones are reviewed as well. Here I will analyze and weigh pros and cons of different intelligence approaches of controlling the PMA.

As was mentioned before, thanks to the obvious advantages of PMA such as high power to weight ratio and muscle like behavior, it is considered an appropriate and safe actuator to use in devices operating in human proximity compared to electric or hydraulic actuators. Recently, in various papers PMA were described as a suitable alternative to hydraulic and electric actuators in medical and rehabilitation robot applications. Commonly used tools such as analytical and numerical approaches are appropriate only for modeling a nonlinear system that is time independent. On the other hand, time varying nonlinear system characteristics can be best modeled using artificial intelligence-based regression models when training data is available. That is why the very first method I will analyze is the artificial intelligence approach. In paper [1], authors tried to accurately predict the uncertain and nonlinear characteristics of PMA using Artificial Intelligence (AI). In this research they made an attempt to analyze Mamdani Fuzzy Inference System (FIS) and Takagi-Sugeno (TS)-based fuzzy systems by means of analyzing the time series data obtained from a real system. They adjusted and tuned these models to achieve higher accuracy and analyzed their parameters, which were tuned using backpropagation through time algorithm (see section 3.4.4) where fuzzy parameters were tuned using three different methods: gradient descent method (GD), genetic algorithms (GA) and Modified Genetic Algorithm (MGA) [1]. They obtained the result that showed that the Takagi-Sugeno fuzzy inference system tuned by Modified Genetic Algorithm gives better accuracy and can also model the time dependent behavior of PMA. The designed TS fuzzy system was found to achieve better results in terms of precision and high deflection when compared to the previous methods in the past. From that one can conclude that the analytical and numerical approaches cannot entirely predict the muscle behavior because most of the research on PMA modeling had been performed for constant loading, and PMA has not been dealing with varying loads, that’s why there are numerous features that have to be taken into account.

AI methods can also be used to deal with a higher degree of uncertainty and ambiguity, which appears as one of their advantageous characteristics as well. Fuzzy logic and ANN are the two well-known approaches generally suitable to establish a mapping between inputs and their associated outputs. Initially, there are two architectures used for ANN modeling, namely, feed forward network and recurrent networks (I will get to them more closely later on). It has been said in the literature that the recurrent networks have restricted performance compared to the feed forward architect of ANN [5]**,** because the feedback loop of a recurrent network passes the data back and forth thus sometimes causes the instability of the system. That is why authors used a multilayered feed forward ANN as a tolerable compromise. Similarly, they described fuzzy logic system approach because of its better accuracy [5]. Speaking more in details, their multilayered feed forward networks were made up of three (or more during testing) layers consisting of input, output and one (or more) hidden layers. Algebraic computations were performed at the hidden layer before passing the inputs to the output layer. As always identification of correct weight matrices was the key to the accuracy of an ANN. In order to identify weight matrices and other key parameters of ANN, researchers used various learning methods such as Reinforced learning, Hebbian learning, Stochastic learning, Gradient Descent (GD) learning, etc. Eventually results obtained from the ANN and the fuzzy models were plotted and analyzed; the fuzzy model approach was found to be more accurate, because it could accurately predict the mapping between force, length, change in length and the pressure inside PMA.

In another paper [6] authors developed an in-house pleated PMA that showed improved response time with rather low hysteresis. In order to deal with the non-linearity and transient nature of pneumatic muscle actuators, authors also proposed an Artificial Neural Network based approach. They came up with a so-called hybrid approach where they combined back propagation (BP) algorithm with Modified Genetic Algorithm (MGA) in order to optimize ANN model parameters. Results they obtained showed that the hybrid approach was able to model the PMA behavior closely; experiments they carried out validated the proposed ANN model and confirmed the advantage of the applied method. Simultaneously authors proposed an artificial neural network approach to model the non-linear indigenously developed pleated pneumatic muscle actuators (PPMA). The trained ANN was able to represent the relationship between force, length, and pressure of the PPMA. The results obtained from the hybrid approach were analyzed in terms of their maximum deviation and convergence rate. It was found that the model developed in this manner could also accurately predict the mapping between force, length, change in length and the pressure inside PMA.

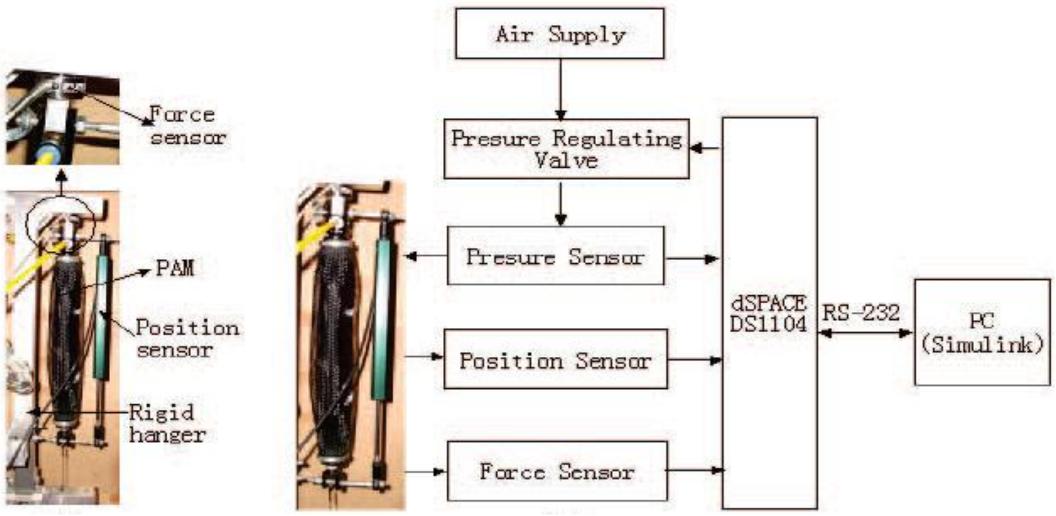

Fig.4 The overall system configuration used for the PMA and PPMA testing, adopted from [6]

In paper [7] authors were engaged with the control of a 3-DOF robot arm actuated by pneumatic artificial muscles. Since the model is intensely non-linear, it was difficult to predict its behavior, so that they needed a stable and robust controlling method. In order to reduce the effects of nonlinearities and uncertainties, authors proposed a combined execution strategy based on neural network and the concept of sliding mode control (SMC). Dealing with this control structure, they used a simple two-layer feed forward neural network with online adaptive learning technique in order to analyze uncertain dynamics and eliminate the so-called “chattering phenomenon” in common SMC [7], and the algorithm was derived from Lyapunov stability analysis.

The execution of Neuro-Sliding Mode technique showed that the trajectory following ability is good; the tracking errors converged to small values less than 0.04 degrees as it was required by the studies of stability analysis and the convergence time was less than 2 seconds (see Table 1) [7]. The algorithm showed good performance when minimizing the nonlinearities in the robot system, and it was also trainable at improvement by supplying neural networks with more input signals in spite of robot nonlinearity and uncertainties caused by system dynamics. Thanks to the property of neural networks as universal approximators, authors were able to work with a two layer neural network in order to reconstruct unknown and unmodelled robot dynamics. It was shown through experiments that the proposed method has a good control performance for the highly nonlinear system, such as the PMA manipulator.

A muscle model analysis was split into two main parts, mechanical and pneumatic.

Table 1 Comparison of Neuro-Sliding mode using NN and conventional Sliding mode control as shown in [7]

	Neuro-Sliding Mode [7]	Typical Sliding Mode Control

Response time	Joint 1: 1.5 s Joint 2: 1.5 s	Joint 1:3 s Joint 2:4.5 s
“Chattering”	Insignificant	Significant
Static Error	Joint 1: 0.02 degree Joint 2: 0.04 degree	Joint 1: 0.1 degree Joint 2: 0.4 degree

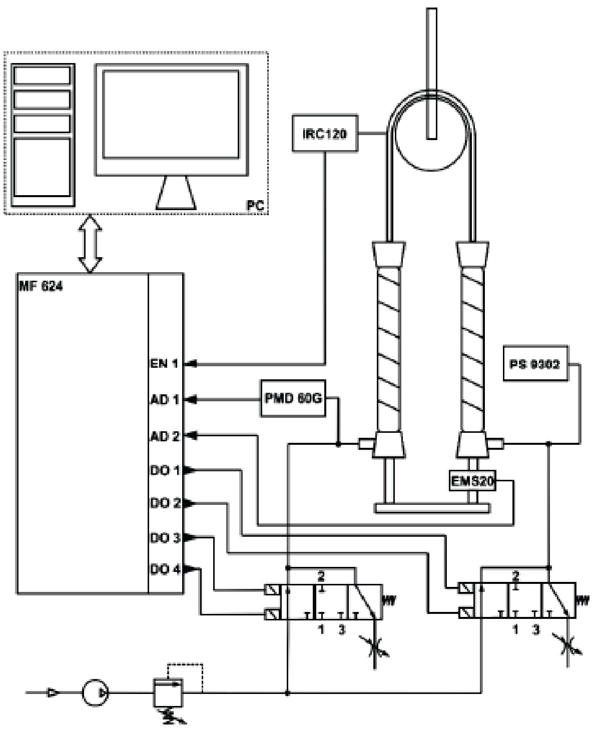

Fig.5 The schematic diagram of experimental setup, adopted from [8]

The general description of the model of the type of pneumatic muscle actuator is given in [8][9], authors came up with the illustrated above experimental setup.

The main problem of authors’ work was to come up with a model of one degree-of-freedom pneumatic artificial muscle-based actuator used for applications in industry (see Fig.5). This model was supposed to be a plant model defining its dynamics that could be used for the control system design using simulation. The aim behind the research was the design of control system for low-cost PMA-based system (the exploitation of on/off valves instead of proportional ones) using desirable nonlinear control techniques capable of dealing with the system’s nonlinear and hysteretic properties. The chosen method was to create an analytical model using so-called grey-box modelling (i.e. modelling using the combination of first principle modelling and experimental parameterization of the model). (More detailed information on what parameterization they used is thoroughly given in the source [8]) It turned out that there was a certain advantage in taking this method as a working one compared to a pure blackbox modelling (e.g. using neuro/fuzzy modeling techniques) in obtaining a model with clear physical interpretation (concerning the modelled physical laws as well as the model parameters) which implies its integrity suitable for modelling an actuator with muscles with different parameters. However, the limitations of this method are also evident when the obtained results were analyzed. Previously mentioned simplifications in modelling resulted in a model with higher valued errors that would probably have been most likely the case for neuro/fuzzy techniques. It is evident that this model could not be used as a hundred percent suitable model but it is acceptable as a model used for PAM-based industrial design applications. In conclusion authors say that there is still a place for further refinement of the model in order to decrease the modelling errors so that the reliance on robustness of the control technique could be diminished.

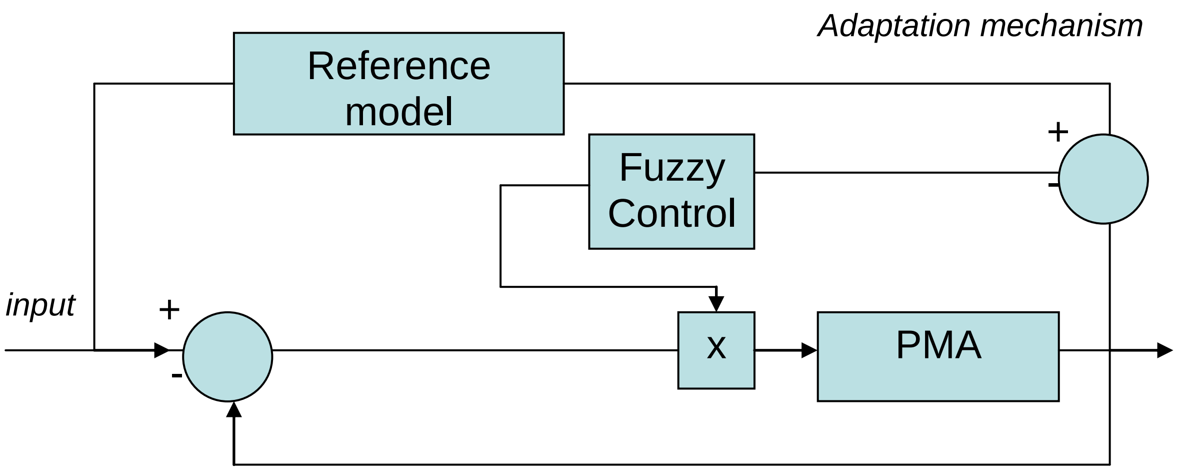

Fig.6 The simplified scheme of the hybrid adaptive fuzzy controller with system adaptation [9]

Particularly in [9] authors proposed a fast hybrid adaptive control method, where they placed a conventional PD controller into the feed forward branch and a fuzzy controller into the so-called ‘adaptation’ branch. The fuzzy controller was supposed to compensate for the actions of the PD controller under chosen conditions (see Fig.6). The fuzzy controller of Takagi–Sugeno type was also modified by means a genetic algorithm using the dynamic model of a pneumatic muscle actuator. The results showed the capability of the designed system to provide robust performance under the conditions of varying inertia. They also confirmed that a fuzzy controller with a simplified fuzzy rule was capable of achieving very good performance (in terms of dynamics errors and uncertainties) even under conditions of a varying inertia moment. The signal adaptation also proved to be fast in achieving improved performance.

3 Used Approaches

In this section I will review and describe the possible approaches of approximating the model of PMA.

The general scheme of our recurrent neural network model is depicted in Fig. 7. General input vector **x** is defined (as in [10]) as follows:

$$\mathbf{x} = \left[y_n(k) \quad y_n(k-1) \quad \dots \quad y_n(k-n_y+1) \quad u(k) \quad u(k-1) \quad \dots \quad u(k-n_u+1) \right]^T \tag{1}$$

where $u(k)$ denotes measured control input and $y_n(k)$ stands for neural output (i.e. neural model of controlled variable) both at reference time index k of a constant sampling.

For neural network models later in this work, however, the augmented vector **x** will be used as:

$$\mathbf{x} = \begin{bmatrix} x_0 = 1 \\ \mathbf{x} \end{bmatrix}. \tag{2}$$

The main benefit of using this model is that it is able to make more accurate long-term predictions under similar conditions in comparison with the typical feedforward model, the training approach in this model, which is consistent with either Levenberg-Marquardt algorithm or a variation of Backpropagation Through Time [10], is built on the trials and error correction learning rules and starts the training process using random initial weights.

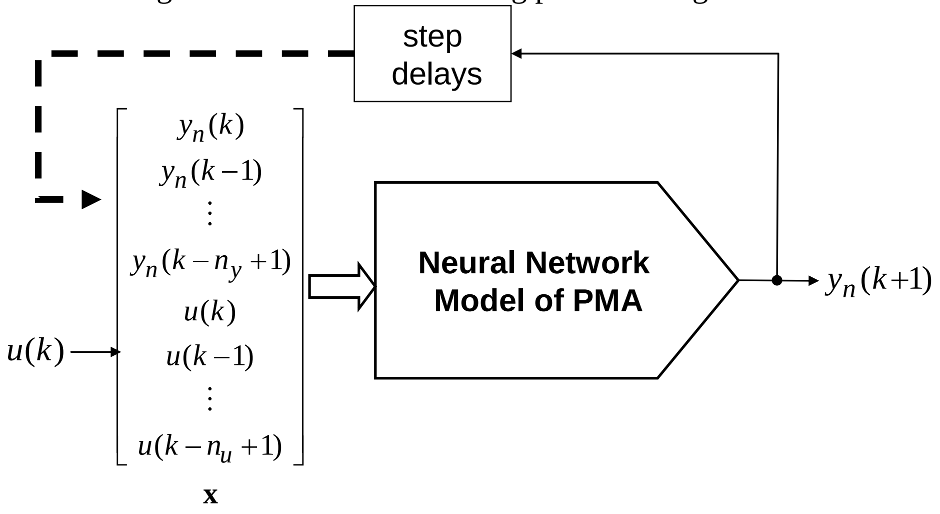

Fig. 7: Principle scheme of recurrent neural network model for PMA.

After determining the output of the model for the input presented in the training set, the error resulting from the difference between the model output and the expected values is calculated and it drives the system to keep on going until the finite number of training epochs is reached.

3.1 Dynamic Linear Neural Unit

The architecture of the DLNU used in this study is shown in the following equation:

$$y(k+1)=\phi(\mathbf{w}\cdot\mathbf{x}) \quad \textbf{(3)}$$

where **w** vector of neural weight as follows

$$\mathbf{w}=\begin{bmatrix} w_0 & w_1 & w_2 & \cdots & w_{1+n_u+n_y} \end{bmatrix}, \quad \textbf{(4)}$$

Where the meaning of n_u and n_y is classified in Fig. 7 and sigmoid function $\phi(v)$ is given as follows:

$$\phi(v)=\frac{1}{1+e^{-v}}. \quad \textbf{(5)}$$

3.2 Dynamic Quadratic Neural Unit

Recurrent QNU [10][11][14] is shown in the following equation where y_n $(k+n_1+1)$ is the neural output (predicted value), W is upper triangular weight matrix augmented with neural bias $w_{0,0}$, n is the number of internal neural recurrent feedbacks, y_r stands for real measured value, n_r is the number of real values feeding the neural unit, k is index of discrete time, and T stands for vector or matrix transposition:

$$y_n(k+n_1+1)=\phi(\sum_{i=0}^{1+n_1+n_r}\sum_{j=0}^{1+n_1+n_r} w_{ij}\cdot x_i\cdot x_j)=\phi(\mathbf{x}^T\cdot\mathbf{W}\cdot\mathbf{x}) \quad \textbf{(6)}$$

$$\mathbf{x}=[y_n(k+n_1) \quad y_n(k+n_1-1) \quad \ldots \quad y_n(k+1) \quad y_r(k) \quad y_r(k-n_r+1)]^T \quad \textbf{(7)}$$

$$\mathbf{W} = \begin{bmatrix} w_{0,0} & w_{0,0} & w_{0,1+n_1+n_r} \\ 0 & w_{1,1} & w_{1,1+n_1+n_r} \\ 0 & 0 & w_{1+n_1+n_r,1+n_1+n_r} \end{bmatrix}. \quad \textbf{(8)}$$

3.3 *Dynamic MLP networks*

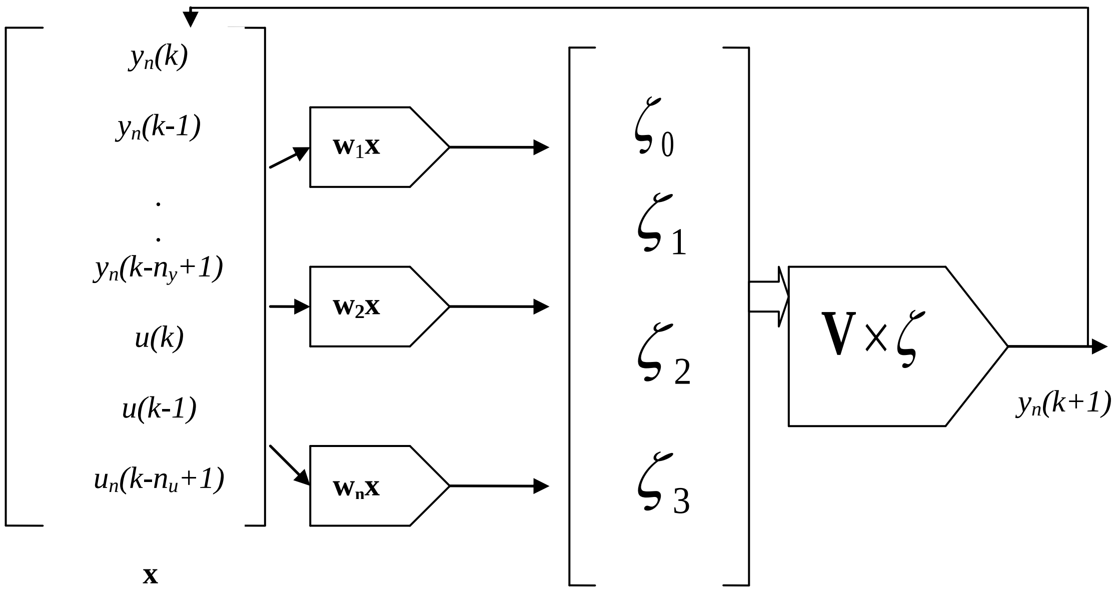

Fig.8 Dynamic MLP structure with one hidden layer

The dynamic MLP network consists of layers of units, each of which carries out a relatively simple operation on its own inputs [12]. The very first layer is made up out of one node for each of the components of the dynamic input data. The neurons in this layer have a transfer function of unity, and their only goal is to spread the inputs to the units in the second layer. The outputs of first layer are connected to the inputs of the second layer. The outputs of the second layer are connected respectively to the inputs of the third layer, and so on. The final layer generates the output values of the MLP. In this manner, the layers between the first and last are not observable from outside the network, and are hence referred to as 'hidden layers'. The connections of the outputs of a layer to the inputs of the next layer have a weight mapped with them. The unit outputs are multiplied by these weights before coming to the inputs of the next layer.

The complete characterization of a MLP demands a number of parameters. These include the number of layers, the number of nodes in each layer, the transfer function used by the nodes, and all the connection weights. Frankly there is no theoretical method for determining the optimum number of hidden layers; however, typically one hidden layer is sufficient for most practical applications as we will see later on. An example of the used dynamic MLP is shown in Fig.8. As for the number of nodes in each layer, the number of input layer nodes is equal to the dimension of the input vector. The number of output nodes is usually determined by the application. There is no conventional way to figure out the optimum number of hidden units; the usual practice is to use a trial and error approach, as that will be shown later. The computation performed by a node on its inputs is referred to as activation or somatic function. Out of the several activation functions that have been proposed, the most popular is the logistic or sigmoid function (5). Typically, a threshold term is included in **x**. This threshold is accounted for by augmenting the number of nodes in the input and hidden layers with an additional node that has a constant output of 1. The function mapping ability of the MLP can be used to either learn the relationship between a set of real-valued inputs and real-valued outputs or binary-valued outputs. In the former case, the MLP functions as a multi-class classifier [12]. In the training batch, the desired outputs for a given input are all agreed to be set to zero, except for the node that corresponds to the correct category of the input. This node receives a desired output of one. After the network is trained and put into operation, the output node with the highest value (and perhaps also satisfying some threshold condition) is declared to be the class to which the input vector belongs. The connection weights are determined using a training algorithm.

3.4 Learning algorithms

There are various types of algorithms for training the neural network. Basically, the purpose of every algorithm is to estimate the local error at each neuron and systematically update the network weights. In this study, the neural networks were trained with the standard Levenberg-Marquardt (L-M) algorithm in case of static NN, and a variation of Backpropagation Through Time (BPTT)[10] in case of dynamic NN to estimate/assess their search efficiency and accuracy in the application for pneumatic muscle actuators. The details on the most common algorithms are given in the following paragraph.

3.4.1 Gradient Descent Adaptation

Fundamental gradient descent rule is as follows:
Considering a static model:

$$y_n(k) = f(\mathbf{x}, \mathbf{w}) \tag{9}$$

with an error:

$$e(k) = y_{real}(k) - y_n(k)\,, \tag{10}$$

one gets weight increment derived from gradient descent formula

$$\Delta w_i = -\frac{1}{2}\mu \frac{\partial e(k)^2}{\partial w_i} = -\mu e(k) \frac{\partial y_n(k)}{\partial w_i} \tag{11}$$

and weight update:

$$w_i = w_i + \Delta w_i \tag{12}$$

3.4.2 Levenberg- Marquardt Batch Training

The Levenberg–Marquardt algorithm (LMA) is a variation of the Newton’s method and it gives a numerical solution to the problem of minimizing a function, generally nonlinear, over a space of parameters of the function. This algorithm calculates weight updates taking into account all obtained data at once (batch training).

Basic formula:

For a single weight one can avoid inverse matrix [13]:

$$\Delta w_i = \frac{J_i^T \cdot e}{(J_i \cdot J_i + \frac{1}{\mu})} \tag{13}$$

where *e*-column vector of all errors, J_i- i-th column of Jacobian.

For multiple weights the use of inverse matrix is needed:

$$\Delta \mathbf{w} = (\mathbf{J}^{\mathbf{T}} \cdot \mathbf{J} + \frac{1}{\mu} \cdot \mathbf{I})^{-1} \mathbf{J}^{\mathbf{T}} \cdot \mathbf{e} \tag{14}$$

In order to train the neural network with L-M algorithm the following steps have to be taken:

1	Define initial values for weights and a learning rate
2	Insert data of all inputs to the network and calculate the corresponding network outputs and errors. Compute the sum of squares of errors over all inputs of ***e***.
3	Compute Jacobian matrix
4	Calculate $\mathbf{\Delta w}$ and update weights

3.4.3 RTRL

Real-time recurrent learning (RTRL) [16] is a gradient-descent based method which calculates the error gradient vector at every time step. It is therefore appears appropriate for online learning applications [19]. The effect of weight change on the network behavior can be noticed by simply differentiating the network dynamics by its weights.

Basic RTRL learning rule for discrete dynamic neural networks [17][11][10] is based on application of gradient descent learning rule (11) to recurrent adaptive (neural) models, where partial derivatives of neural inputs $\frac{\partial y_n(k+\mathrm{n}_s)}{\partial w_{ij}}$ are recurrently calculated as auxiliary recurrently evolving variables, as it is indicated by the following example of Jacobian vector for weight w_{ij} as

$$j_{ij} = \frac{\partial \mathbf{x}}{\partial w_{ij}} = [0 \frac{\partial y_n(\mathrm{k}+\mathrm{n}_s-1)}{\partial w_{ij}} \frac{\partial y_n(\mathrm{k}+\mathrm{n}_s-2)}{\partial w_{ij}} \dots \frac{\partial y_n(\mathrm{k}+1)}{\partial w_{ij}} 0 \dots 0]^T . \qquad \mathbf{(15)}$$

Regarding properties of RTRL, here we may cite from [19]: “high computational cost makes RTRL useful for online adaptation only when very small networks suffice.”

However, the solved problem in this thesis uses still small network.

3.4.4 A variation of BPTT

A variation of Back propagation through time (BPTT) algorithm [10] can be implemented as a batch training gradient-based technique for training recurrent neural networks. It can be described as combination of Gradient Descent and Levenberg-Marquardt algorithm:

$$y_n = w_{0,0} + w_{0,1}x_1 + w_{0,2}x_2 + w_{1,1}x_1^2 + w_{1,2}x_1x_2 + w_{2,2}x_2^2 =$$

$$= [w_{0,0} \quad w_{0,1} \quad w_{0,2} \quad w_{1,1} \quad w_{1,2} \; w_{2,2}] \cdot \begin{bmatrix} x_0^2 \\ x_0x_1 \\ x_0x_2 \\ x_1x_1 \\ x_1x_2 \\ x_2^2 \end{bmatrix} = \mathbf{w} \cdot \mathbf{x} \qquad \mathbf{(16)}$$

Small note concerning RTRL and BPTT: In very recent implementations of LNU and QNU during experimental analyses , it appeared that recurrent derivatives , as indicated in (15), can be neglected and RTRL and BPTT techniques work also fine for identification and control analysis, e.g. [20]. Therefore, the recurrent derivatives are further neglected in programming the codes in this thesis.

3.5 Description of the Used PMA Model

Simulation model of PMA based actuator was realized in Matlab’s Simulink environment [8][9]. The main part of the model PMA [8][9] enhanced with PWM [24] is depicted in Fig.9.

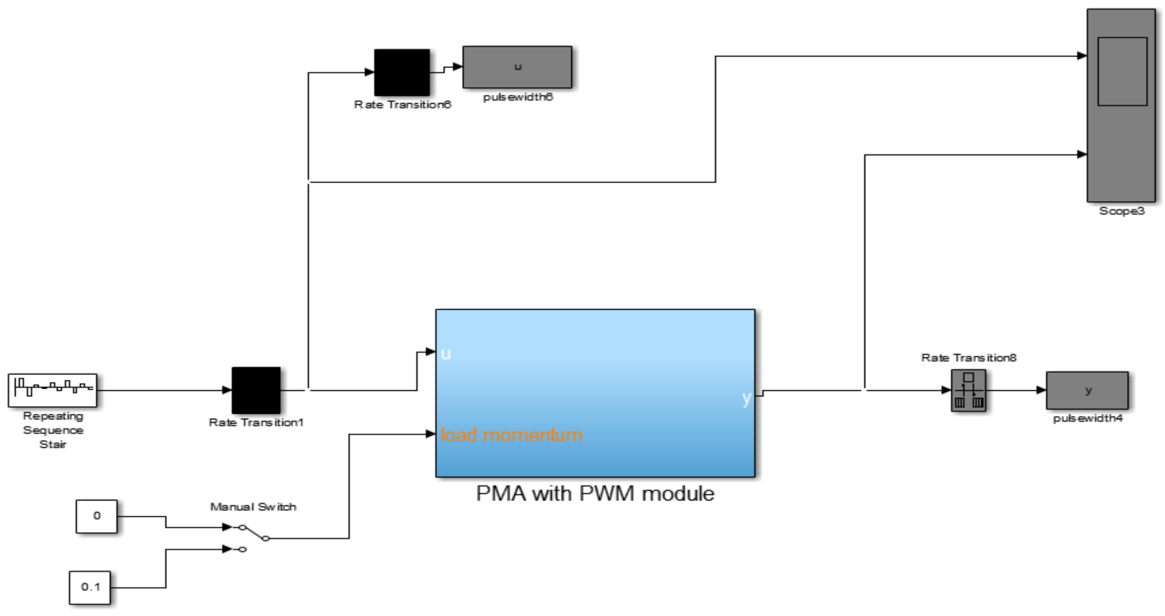

Fig.9 PMA model, adopted from [8][9] extended with PWM

To illustrate a practical application of the techniques described in previous sections, the PMA model is considered to control pressure input by changing the valve position from -1 to 1 (open/closed).

The subsystem of the PWM looks as follows:

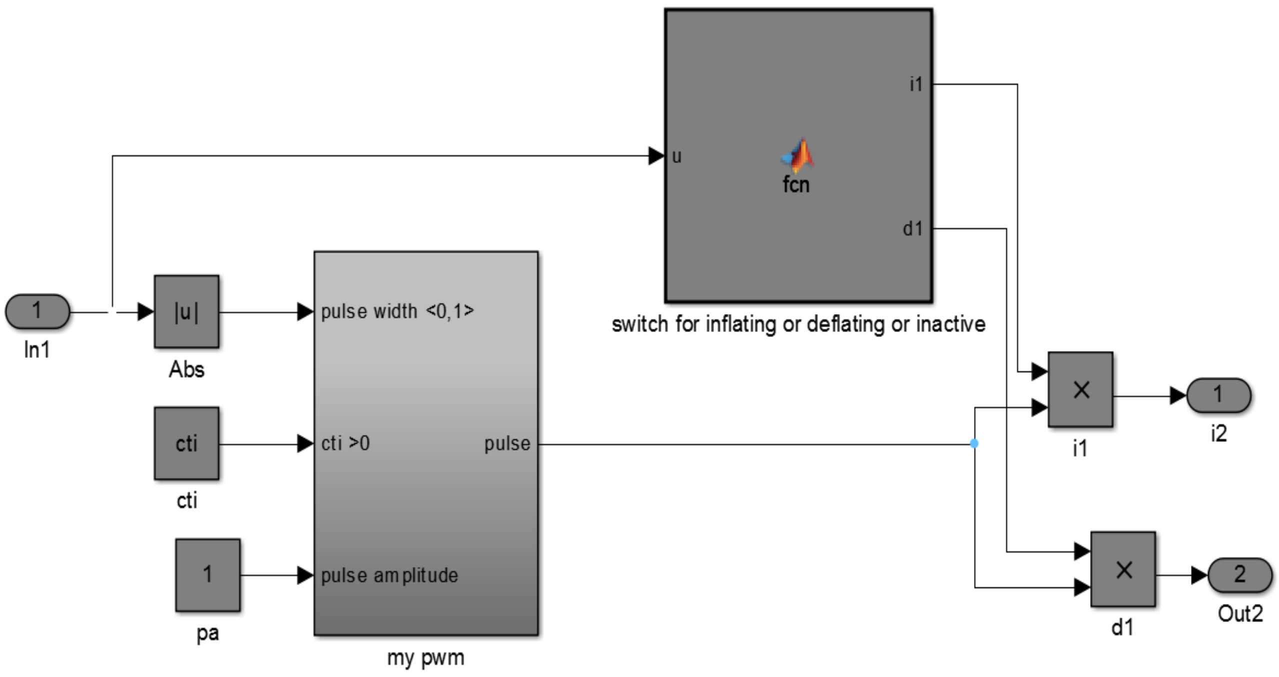

Fig.10 Subsystem of PWM, model [8][9] with PWM extension [24]

Real data obtained from the Simulink model of PMA by means of using PWM is illustrated in Fig.11.

One of the specifics of the given PMA model is that it differs from the classical SISO (single input-single output) system because the pulse-width modulation technique was involved.

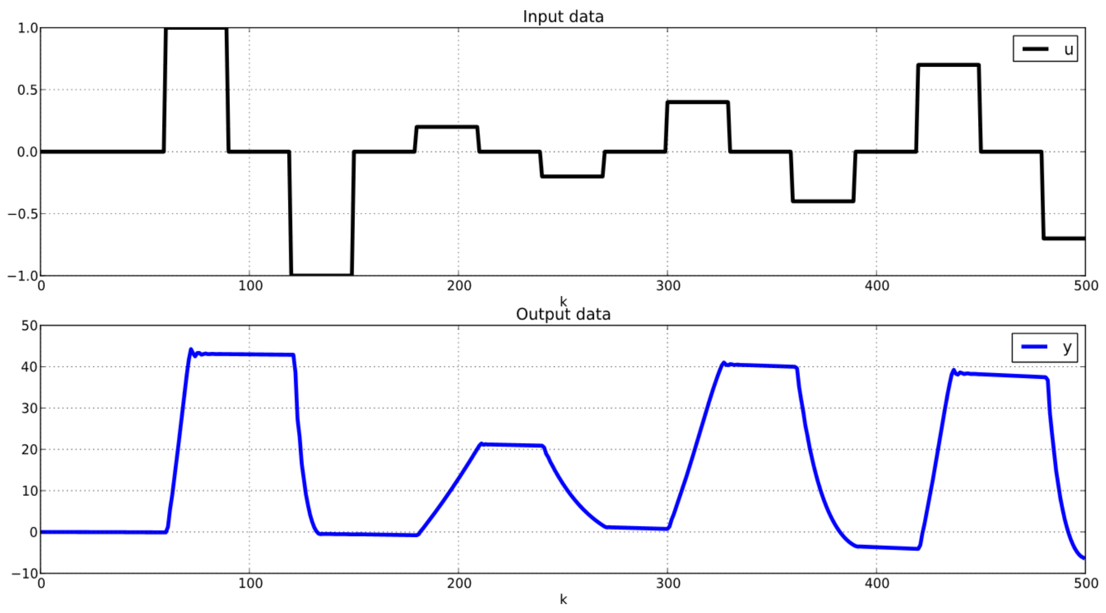

Fig.11 Real data obtained from the model

Classical SISO system looks as follows:

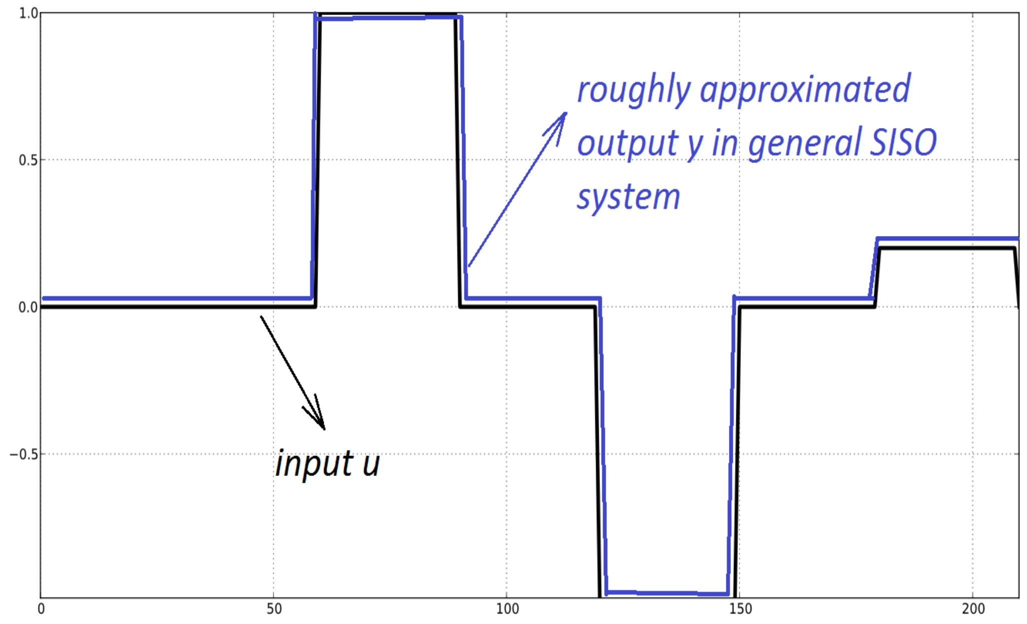

Fig.12 Classic SISO system, the output data usually reflects the input

Our model of PMA[8][9][24], on the other hand, shows that output data doesn’t fully respond to changes in input data due to pulse-width modulation:

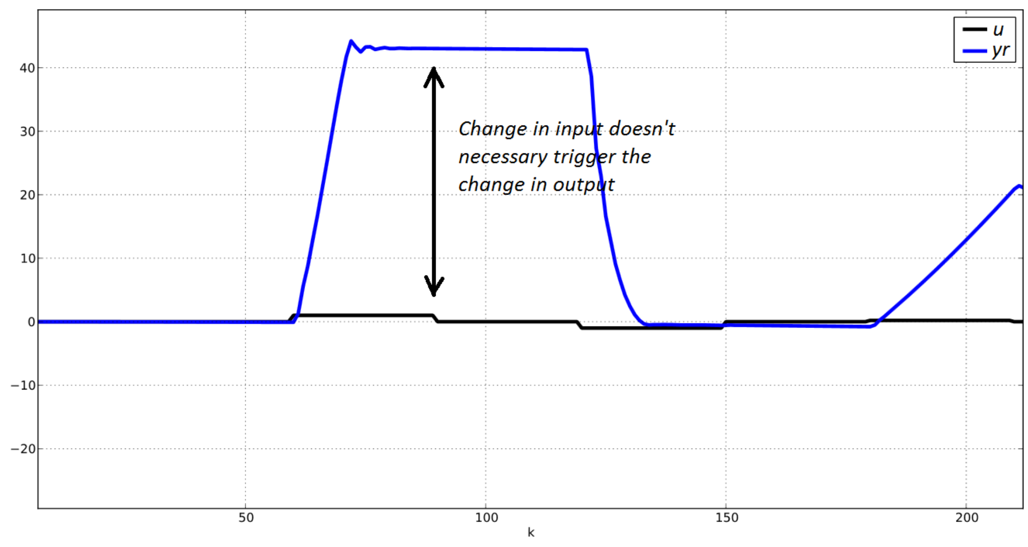

Fig.13 Input-output of our model [8][9][24]

4 Experimental Analysis

In this section I will outline the stages of designing the programs for training and testing of the neural networks explained in the previous chapters. Using the above theory, twelve different programs will be written in Python programming language for training and testing static and dynamic LNU, QNU, and MLP. Furthermore, I will calculate the Sum of Squared Errors and the Coefficient of Determination in order to compare the efficacy of the designed neural networks.

There are many different attributes that have to be taken into consideration when dealing with neural networks, such as learning/training paradigms, network topology, and network function, special features, etc. Commonly used approaches for defining the outlines of selection of the initial setup can be found in [15]. In this case the assessment must include: selecting a common existing structure for which training algorithms are available; adapting an existing structure to suit the application, so that neural network would show better performance [23]. The architectures and learning algorithms of these networks are briefly described in the previous section. Here, in the experimental analysis for approximation purposes I decided to use various types of neural networks since it's the fairest method to identify the most suitable one for this work.

4.1 Used Criteria

As the main identifiers of how well the neural network performs I will take two values that are:

1. The so-called Sum of Squared Errors which is calculated as follows:

$$SSE = \sum_{0}^{N} (y_r - y)^2, \qquad \textbf{(17)}$$

where *yr* is a value from a real data, *y* is a value of a trained neural network that is to be predicted.

SSE is a measure of the deviation between the given data set and a designed model. The smaller the SSE, the better our model fits to the data.

2. The second term that will be used is the Coefficient of Determination, also known as R squared, it shows the 'goodness' of fit of the neural network model. The higher the coefficient of determination, the greater dependence one sees between the real variable and a predicted one. All in all, it acts as a total measure of the usability of a system.[21] It is derived as follows:

$$R^2 = 1 - \frac{\sum_0^N (y_r - y)^2}{\sum_0^N (y_r - \bar{y})^2}, \tag{18}$$

where $\bar{y}$ is a mean value of the renormalized data set.

Author in [22] came up with following outlines concerning the coefficient of determination:

1. A generalized coefficient of determination should be consistent with the classical coefficient of determination when both can be computed;
2. Its value should also be maximized by the maximum likelihood estimation of a model;
3. It should be, at least asymptotically, independent of the sample size;
4. Its interpretation should be the proportion of the variation explained by the model;
5. It should be between 0 and 1, with 0 denoting that model does not explain any variation and 1 denoting that it perfectly explains the observed variation;
6. It should not have any unit.

$R^2 > 0.9$ can be considered as a good overall fit.

Later on, I will calculate these values and compare them in order to analyze the performance and efficiency of each network.

The data consisting of 1000 samples was normalized in order to reduce the noise and avoid inconsistency. During the update procedure I will use the Levenberg-Marquardt algorithm and the Back Propagation Through Time (see the previous section on learning algorithms). The network training process is performed by providing input-output data to each network, which targets minimizing the error by optimizing the network weights and performing the corresponding computations.

Initial setup for Python codes (see full codes in *Appendix*):

n_{yr}---- range of the real output data y in dynamic model system

n_{ur} --- range of input data u in dynamic model system

$n_x = 1 + n_{yr} + n_{ur}$----(vector matrix **x** *- first column in the* Fig. 7*)*

mu (μ) --- learning rate of the neural network, its optimal value was found experimentally for each network

epochs--- number of epochs

w=randn(n_w)/n_w---- we'll set initial weights to some random values in range $n_w = n_x$.

The learning rate and the random weight range are two basic network parameters that significantly affect the performance of the model by changing its weights. At first they can be set at default values and if the model is unstable they can be made smaller until it stabilizes. Later on, one will be able to see that changing the weight range or the learning rate does not necessarily result in large changes in model accuracy. The deviation in precision is approximately in the same range as that of the random start, which initializes the weights. Because of the variation in model performance caused by using different neural networks and learning algorithms respectively, I will run all network configurations at least three or four times using the same predetermined random values, produced by the Python's random number generator. Thus I will optimize the network parameters and network architecture based on the average of the several random starts.

When the training is complete, I also would like to check the network performance and determine if anything needs to be changed to the training process, the network model or the data sets.

At first the learning rate parameter was typically set to 0.01, but was adjusted every once in a while for each network so that the learning rate was decreased if the system become unstable and vice versa.

The 'SSE vs. the iteration number' diagrams were also plotted and the smallest value of SSE was found. The weights derived at the error minimum for the train/test set were selected and considered as the optimal weights, since these give the best generalization properties. It was verified that the size of the training set (1000 samples) was large enough by examining the generalization properties as the number of samples of the training set was varied. Furthermore, in order to emphasize the specific nature of each learning algorithm and to avoid the neural network getting undesirable error values, every training procedure was repeated at least 5 or 6 times and the best fits of the those sessions were used.

Considering the significant importance of the network architecture, in this part before exploiting different types of neural network models, some points related to the network architecture should be mentioned: in order to find the optimal number of neurons, an attempt was made to evaluate different networks with different number of neurons. Therefore, 2 to 15 neurons were used while search of the best fit; each network was trained at least 10 times and in order to compare their performance, the SSE in the test data (which included 50% of the whole data) was also set as the criteria. Finally, the optimal number of neurons was found individually for each one and the optimal number of hidden layers was determined to be 1 because the addition of unnecessary hidden layers can make the network too complex and use more epochs for training.

Frankly, the neural network structure that gives the best possible result can only be determined experimentally, there is no other way to say if the network is capable of solving this task. The quality of a performance of a neural network is strongly dependent on the network parameters such as initial set of weights and learning time, which influences the networks' capabilities to generalize the whole data; generalization is the ability of the neural network to 'understand' and process the data that it has not seen before. Later on I will focus on investigation of how well it learned from the training data and how it handles the brand new sets of values. The size and the characteristics of the training data set together with the number of iterations are the other factors affecting the generalization capabilities of a neural network, so it must be taken into consideration as well.

4.2 Approximation of PMA with Linear Neural Unit

4.2.1 Static LNU Using L-M Algorithm

As in every preliminary theoretical research, the linear neural setting should be considered as the first simple case to be studied. The advantage of the static linear neural unit is that the network can easily be built with a simple optimizing algorithm (e.g. Levenberg-Marquardt). Nonetheless, the static LNU also has several drawbacks for some applications. First, it may fail to produce a satisfactory solution because the training data can require more complex approach. Second, static LNU cannot cope well with major changes that were never learned in the training phase.

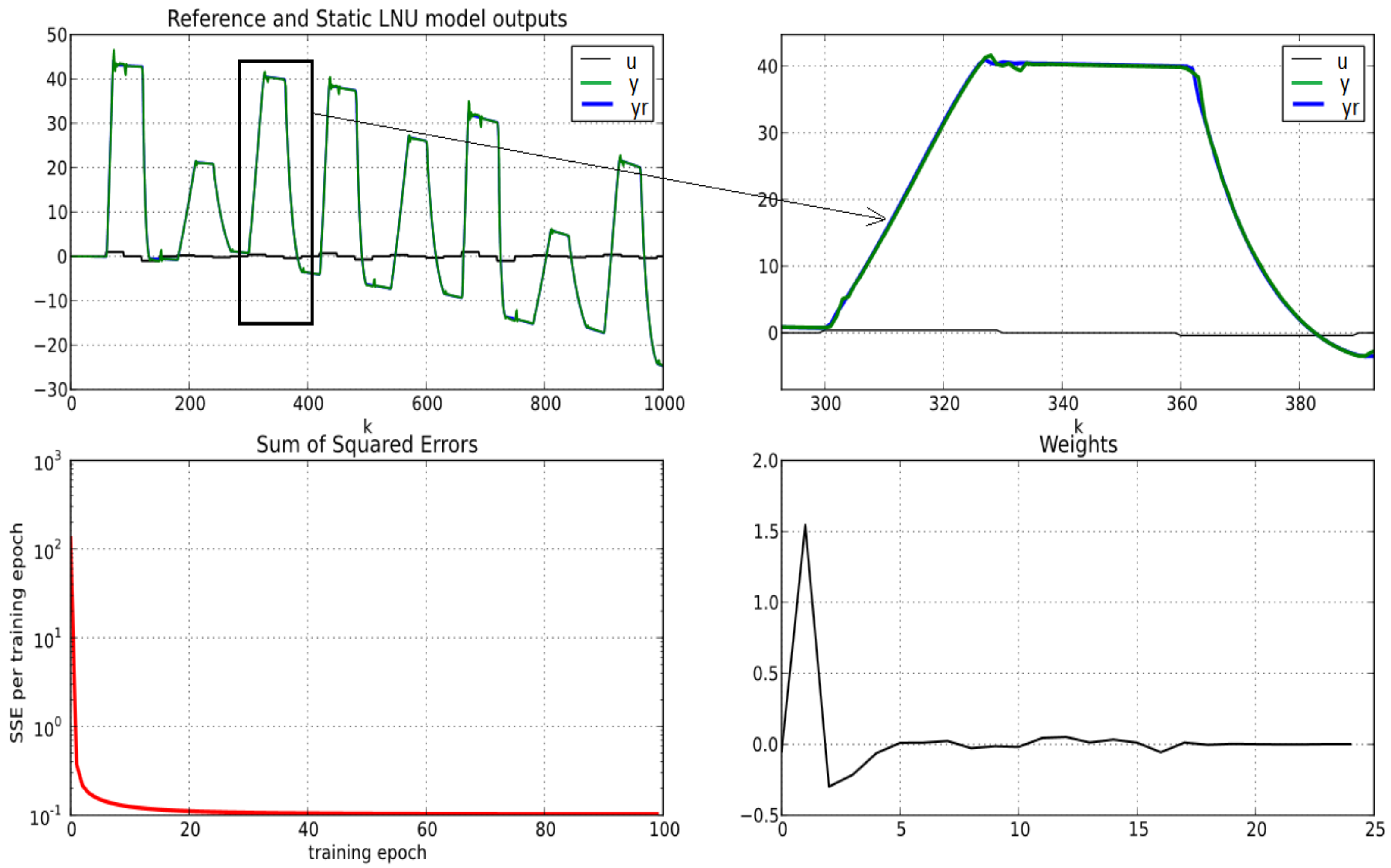

Fig.14 Static LNU trained with L-M Algorithm (nyr=12, nur=12, μ=1, SSE=0.102, sampling=0.1 sec, epochs=100). The values of SSE and R^2 are fine (see Table 2), however, the neural network doesn't seem to learn well due to the specific behavior of our SISO system, it couldn't capture the nonlinear character of the PMA's input and output.

4.2.2 Dynamic LNU Using BPTT Algorithm

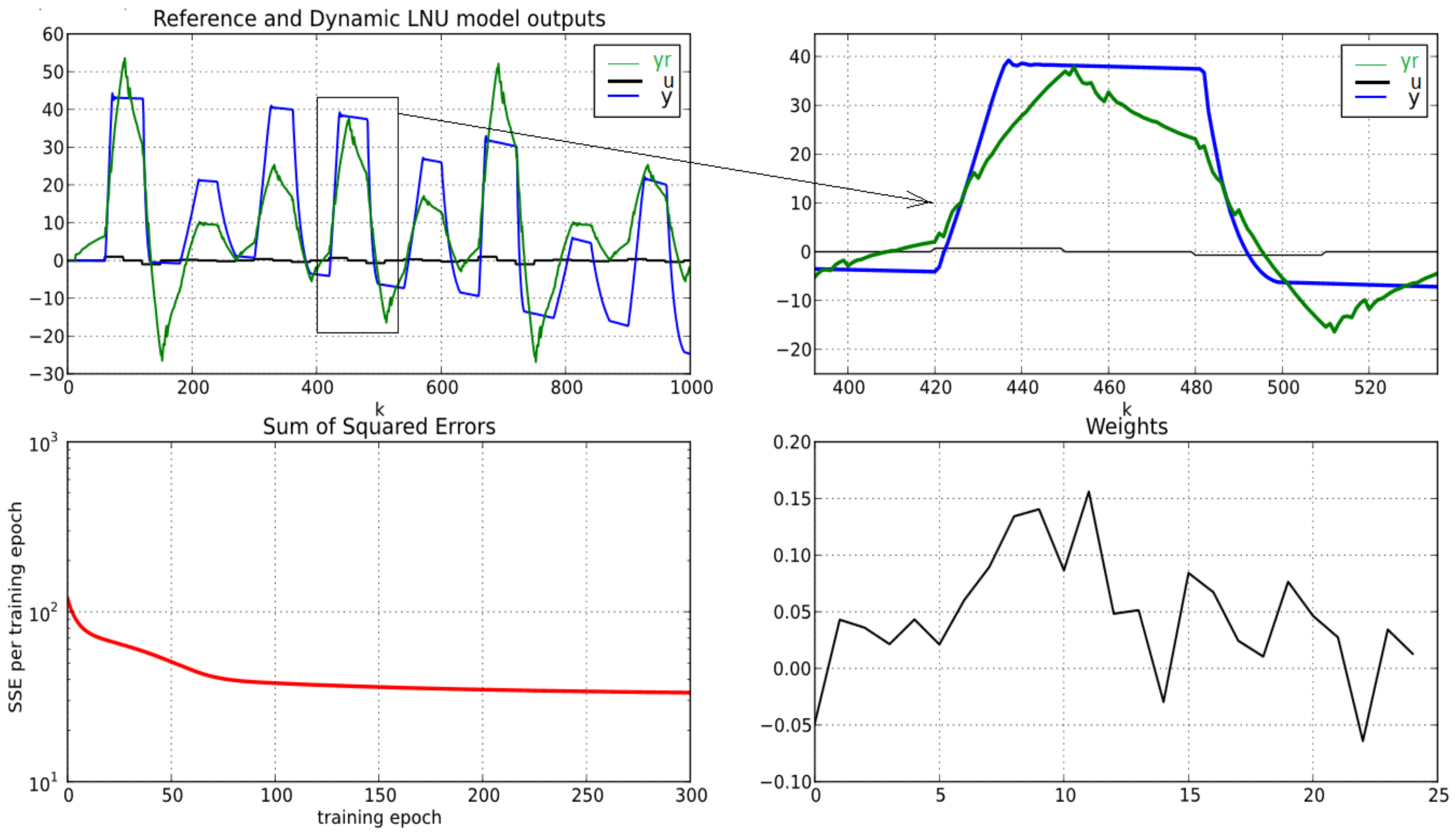

Fig.15 Dynamic LNU trained with BPTT Algorithm (nyr=12, nur=12, μ=0.0001, SSE=33.436, sampling=0.1 sec, epochs=300). As one can notice the values of SSE have increased, although I significantly decreased the learning rate in order to improve stability of the algorithm and the stability of neural network itself. Higher values of learning rate tend to disable the whole system.

4.3 Approximation of PMA with QNU

4.3.1 Static QNU Using L-M Algorithm

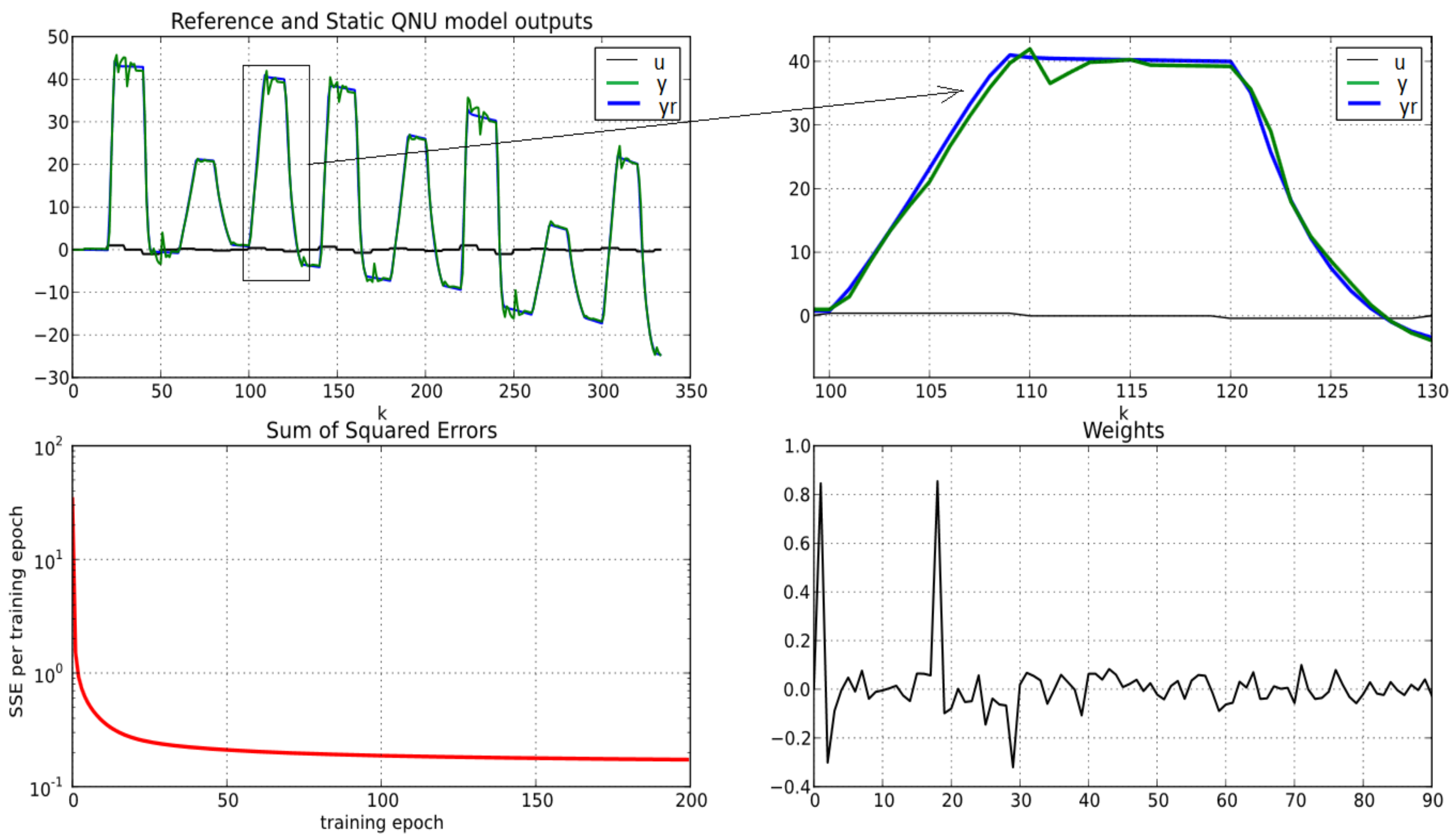

Fig.16 Static QNU using L-M algorithm (nyr=6, nur=6, μ=0.1, SSE=0.164, sampling=0.1 sec, epochs=400).

4.3.2 Dynamic QNU Using BPTT Algorithm

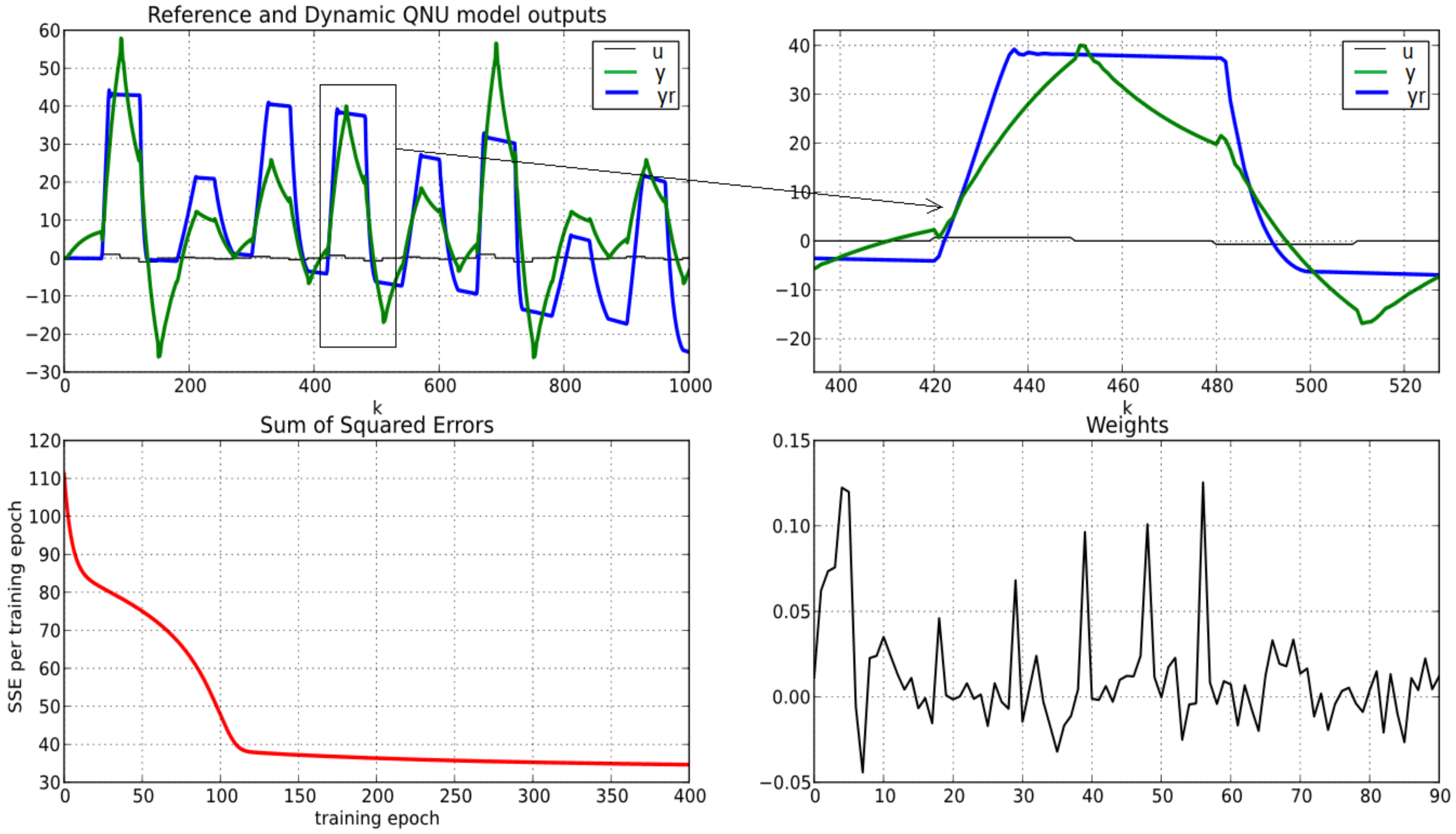

Fig.17 Dynamic QNU using BPTT algorithm (nyr=6, nur=6, μ=0.0001, SSE=36.125, sampling=0.1 sec, epochs=400). I had to choose small learning rate in order to maintain approximation accuracy and tolerable values of SSE and R^2. Also I noticed that the speed of convergence in this case highly depends on the size of training set and several other user-defined parameters such as nyr, nur, etc.

4.4 Approximation of PMA with Static MLP Network

4.4.1 Static MLP Using L-M Algorithm

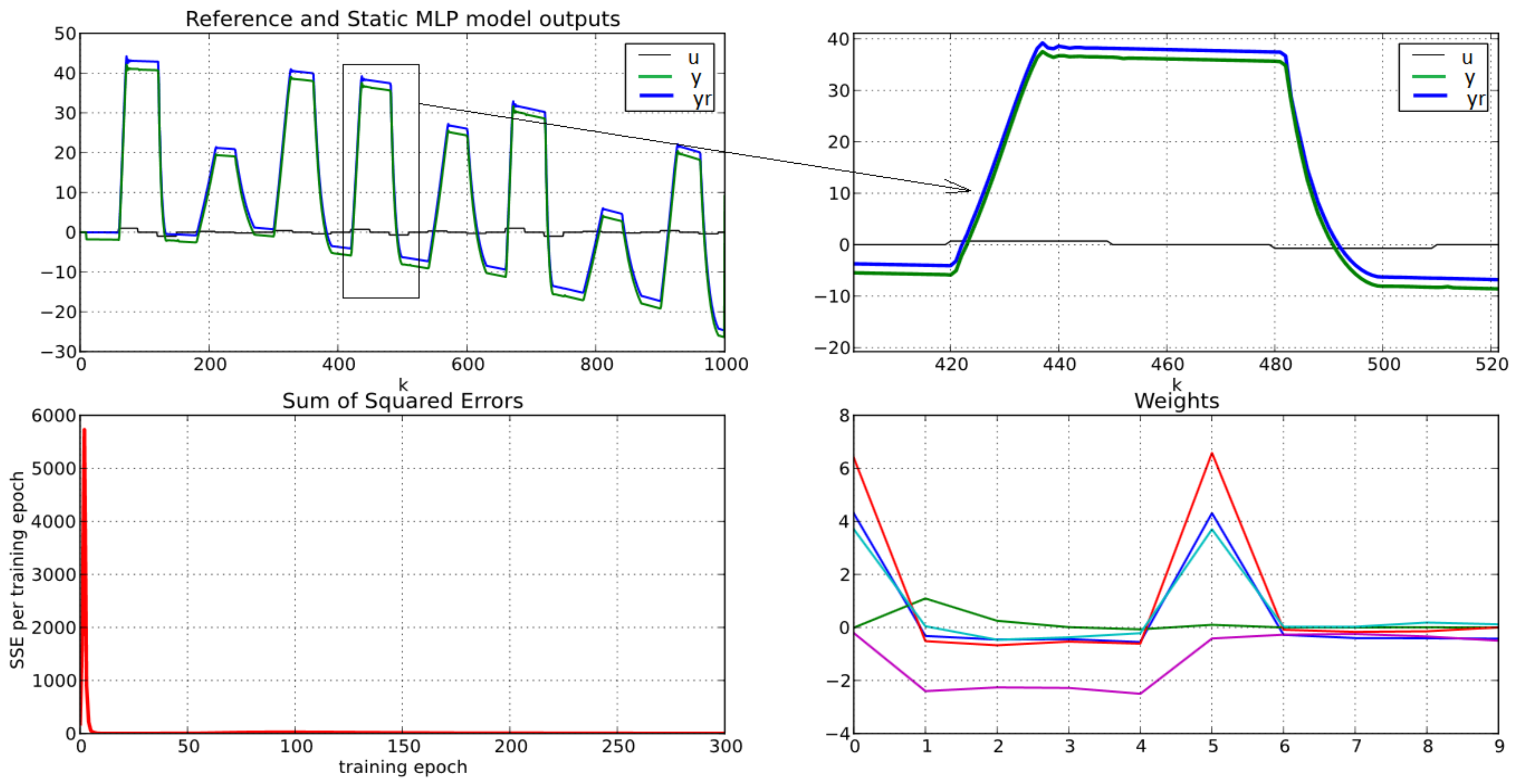

Fig.18 Static MLP using L-M algorithm (nyr=5, nur=10, μW=0.01, μV=0.005, SSE=0.169, sampling=0.1 sec, epochs=600). Here I was trying to adjust the learning rates ranging from 1 to 0.0001 and eventually found the most suitable one that helped me achieve the stable and acceptable result. This neural network showed the best performance in terms of efficiency and values of SSE and R^2. Plus it has a tendency to compensate the specific behavior of the given SISO system (see Fig.13).

4.4.2 Dynamic MLP Network Using BPTT Algorithm

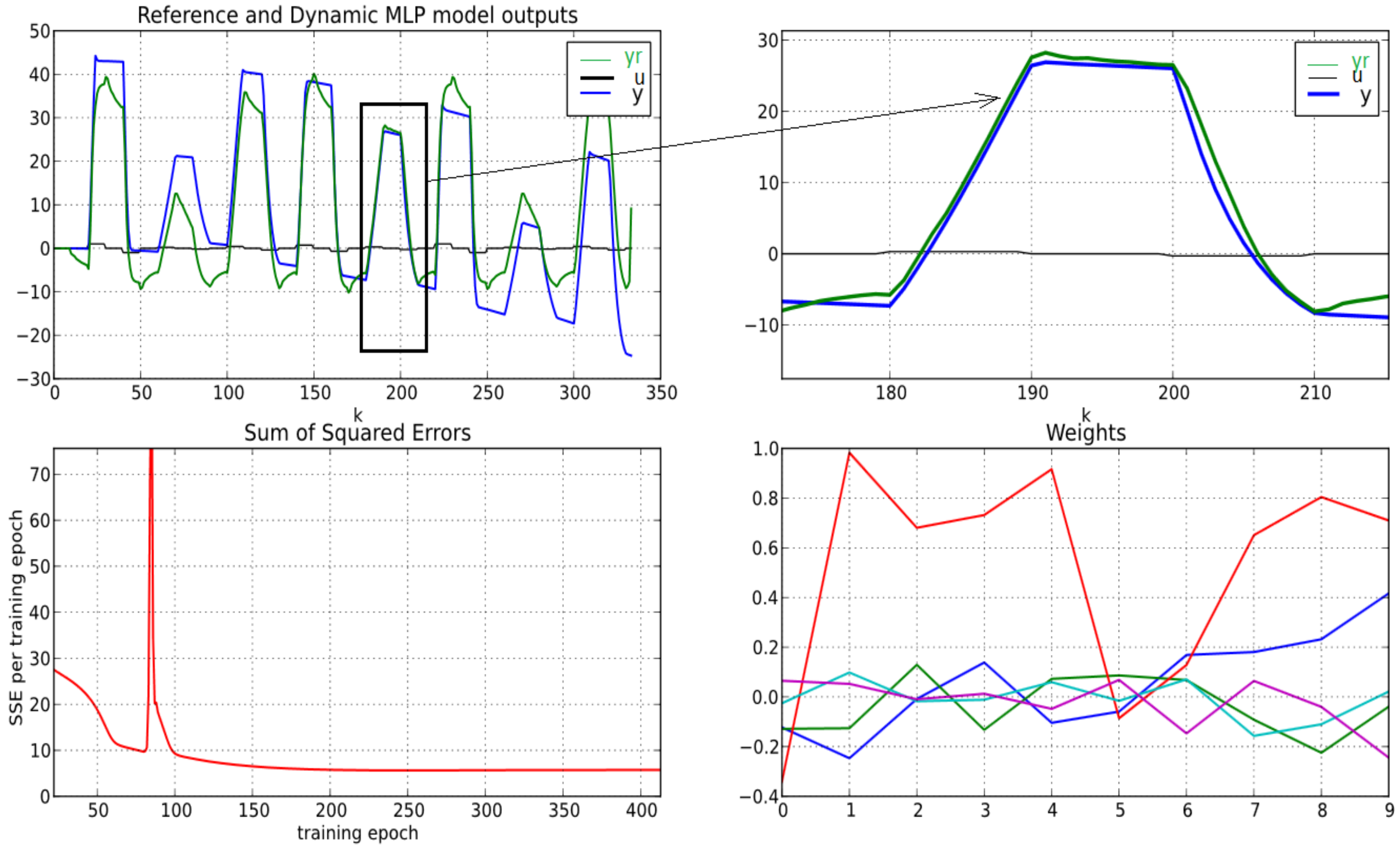

Fig.19 Dynamic MLP network using BPTT algorithm (nyr=5, nur=10, μW=0.0, μV=0.005, SSE=5.653, sampling=0.1 sec, epochs=500). The results demonstrate that the number of neurons in the hidden layer were sufficient for the network to learn the training samples.

4.5 Testing of the Neural Networks

To verify if the neural networks have adequately learned the input domain I will perform the testing procedure with similar data as that used in the training set (last 500 samples). The testing procedure is very important, because one needs to know if the trained neural networks meet the requirements to the solution of approximation of PMA.

The results of the testing of the networks are shown in the following pictures.

Testing of the Static LNU:

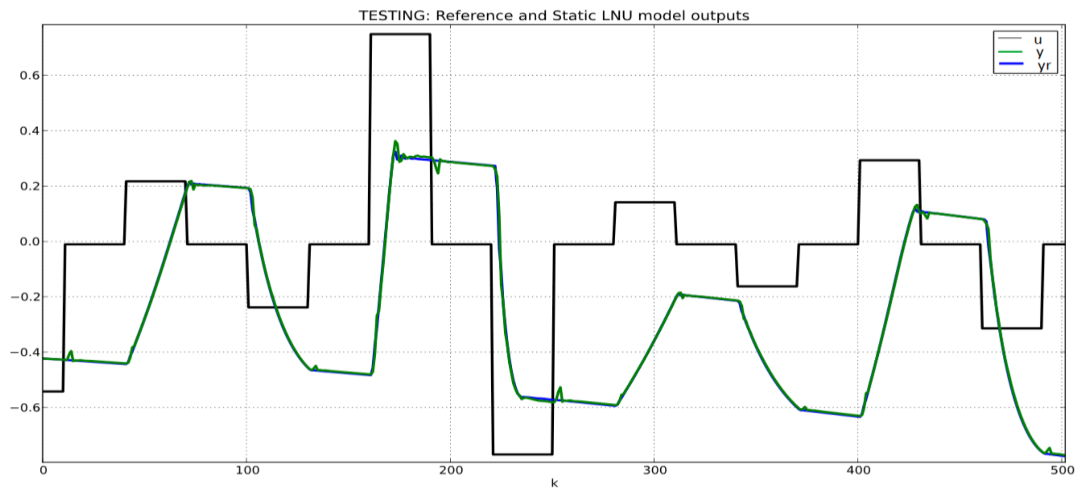

Fig.20 Testing of Static LNU: SSE=0.528, sampling=0.1 sec

When I applied the trained model of LNU on the independent data set I saw that the curve of the trained model was quite similar to the curve of the tested model, the value of R^2 was equal to ~0.99, which is supposed to indicate the close fit of both real and tested data (as a reminder: if $R^2 = 1$, this means that there is a precise linear relationship between outputs of the neural network and the real data. If R^2 is close to zero, then there is no linear relationship whatsoever). However, as in case of training I was still facing the problem of avoiding the specific nature of the input-output mapping of our SISO, even in the testing phase it was not possible to prevent the neural network's output curve from simply following the input curve.

Dynamic LNU:

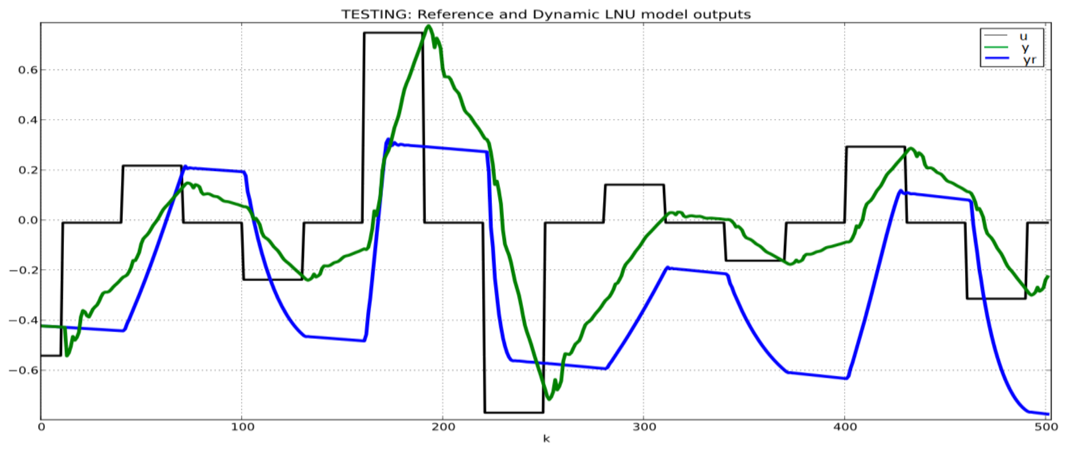

Fig.21 Testing of Dynamic LNU: SSE=38.209, sampling=0.1 sec.

Fig.21 shows the testing results for dynamic LNU, which basically suffers from poor capability for generalization. The slopes are very steep and the testing curve is pretty far from the actual (real) one. The coefficient of determination also was quite small: ~0.23.

Static QNU:

The difference between the training and testing sets in case of the static QNU was smaller; it was trained with smaller learning rate than LNU which resulted in a lower SSE for both the training and testing. Thus, it yielded to the network with slightly better generalization ability.

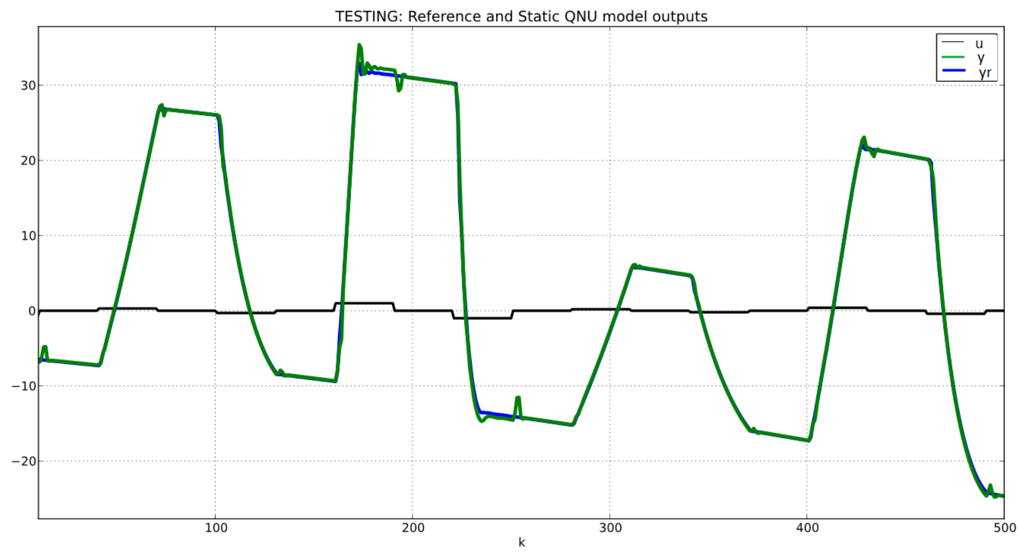

Fig.22 Testing of Static QNU: SSE=3.177, sampling=0.1sec.

Dynamic QNU:

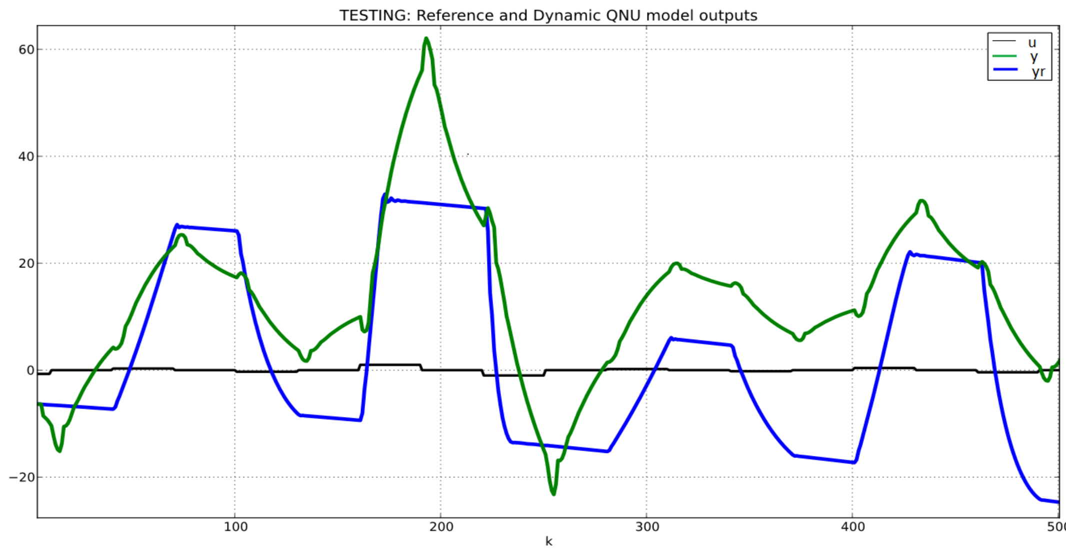

Fig.23 Testing of Dynamic QNU: SSE=38.966, sampling=0.1 sec

As it is seen in Fig.23, the SSE in this case significantly increased compared to the static QNU due to specific features of dynamic QNU, R^2~0.35.

Static MLP: The testing of static MLP showed better generalization capability, and the values of SSE and R^2 in Table 3 indicates the superiority of static MLP over the rest of the tested neural networks, because only this network seemed to manage the behavior of PMA.

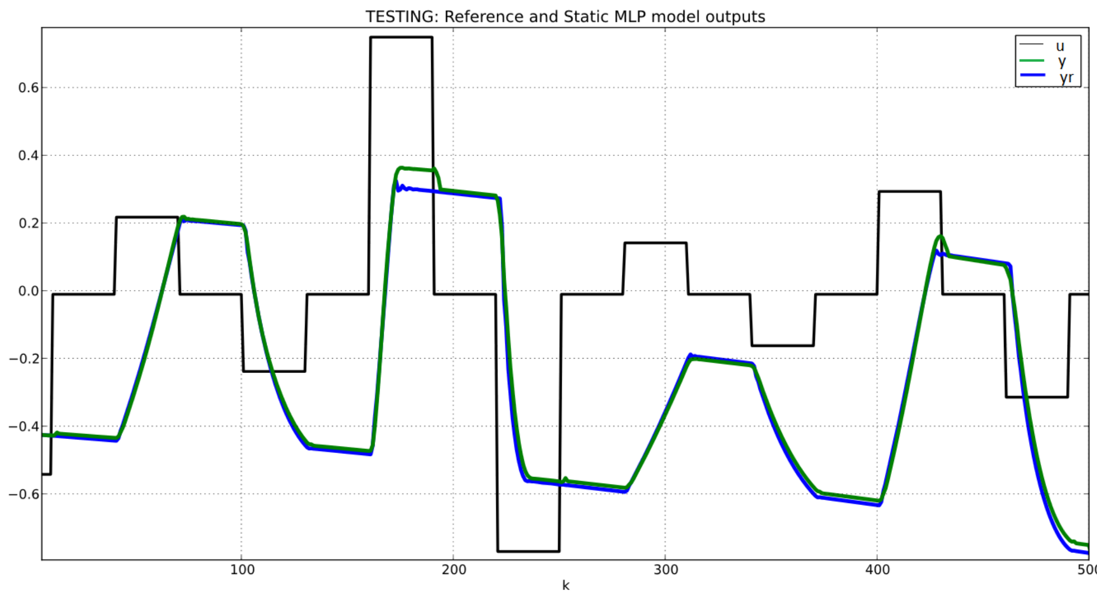

Fig.24 Testing of Static MLP: SSE=5.837, sampling=0.1 sec

Dynamic MLP: one of the main features I noticed about the testing of dynamic MLP is that it takes less time processing the data, although the accuracy of testing was worse than in case if static MLP:

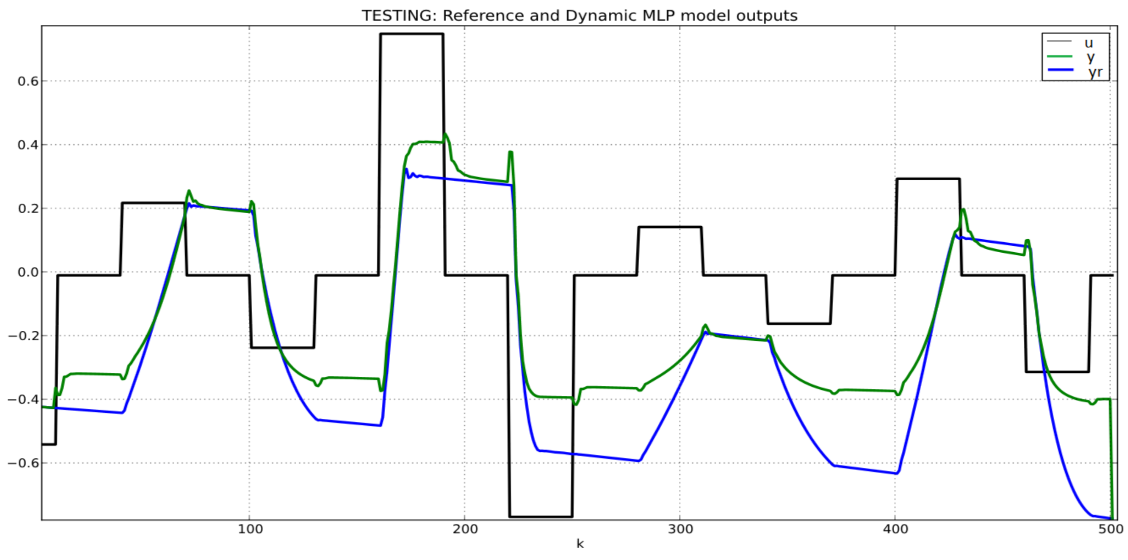

Fig.25 Testing of the Dynamic MLP: SSE=8.864, sampling=0.1 sec

5 Results on Comparison of Neural Networks

In this section I will summarize and conclude which one of the neural networks showed the best ability to learn during training and the best ability to generalize during testing. To do so, I calculated the values of SSE and R^2 that would help to decide whether the neural network was able to capture the characteristic behavior of the model of PMA.

As it was mentioned before, SSE and R^2 are the most commonly used criteria for comparing different models in accurate forecasting of the results. In many studies the SSE criterion is used as a measure of fitting accuracy of models and includes all the features of the other criteria including taking into consideration the outer data and comparing the accuracy of models as well as showing the error differences. Therefore, it was quite reasonable for me to build the assumptions on the basis of the mentioned criteria. The results of comparison have been presented in Table 2 and Table 3. As shown there, the static neural networks in general perform better in comparison with the dynamic neural network models.

Table 2 Coefficients of Determination R^2 and SSE for TRAINING of Neural Architectures

Neural network	Coefficient of determination	Sum of Squared Errors
LNU	0.996	0.102
DLNU	0.294	33.436
QNU	0.993	0.164
DQNU	0.405	36.125
MLP	0.958	0.169
DMLP	0.504	5.653

Table 3 Coefficients of Determination R^2 and SSE for TESTING of Neural Architectures

Neural network	Coefficient of determination	Sum of Squared Errors
LNU	0.994	0.528
DLNU	0.238	38.209
QNU	0.994	3.177
DQNU	0.347	38.966
MLP	0.980	5.837
DMLP	0.986	8.864

Using the obtained values of SSE and R^2 from the tables above I created two diagrams that graphically show those values for each network (Fig.26 and Fig.27).

As one could notice, the values of SSE and R^2 presented above don't necessarily reflect the real situation in the given case, because the majority of used neural network couldn't handle the specific nature of the PMA, which occurred due to the use of pulse-width modulation (see Fig.15, Fig.17, Fig.19).

For example the static LNU (Fig.14) has excellent values of SSE and R^2, but it couldn't apprehend the peculiar behavior of our SISO system, which makes it not a good choice for the approximation procedure.

As for the network that performed better than others, I would say it was the static MLP (Fig.18), because it was able to capture the nonlinear action of the PMA model and practically handle its characteristic nature. The values of SSE and R^2 were also quite good, so that it would make the static MLP by far the most suitable neural network for the given model.

Other neural networks were not capable of dealing with the model, because it turns out they don't perform well in terms of capturing nonlinearities and handling them, so when it comes to the specific nonlinear behavior of PMA they are not the optimal solution for approximating techniques.

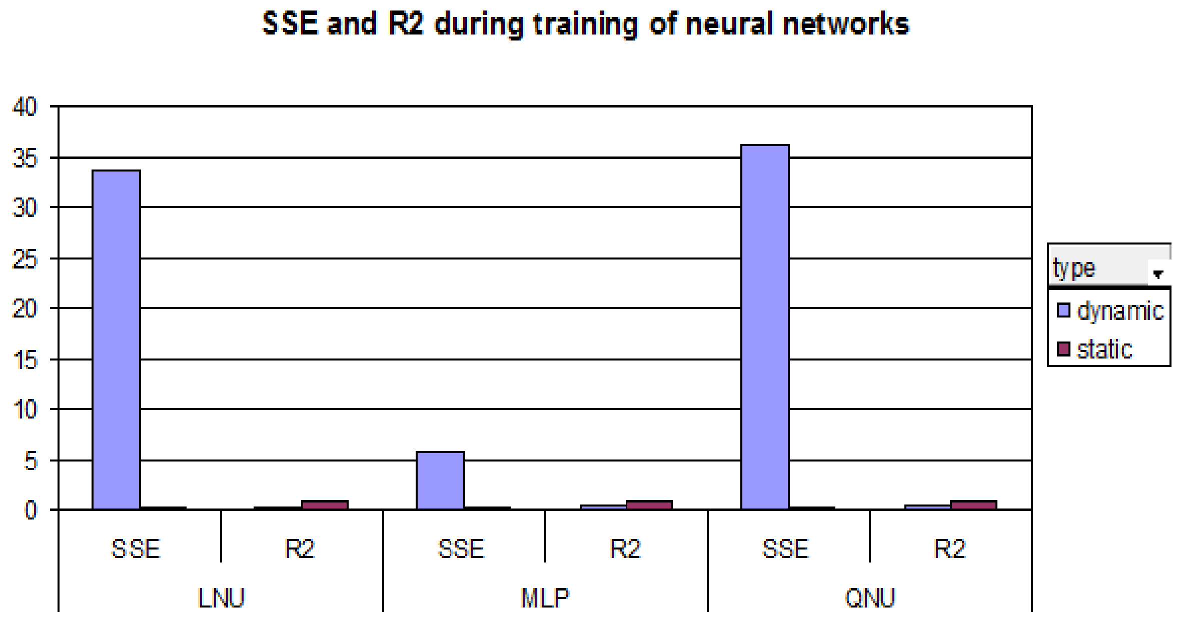

Fig.26 SSE and R^2 during training

During testing:

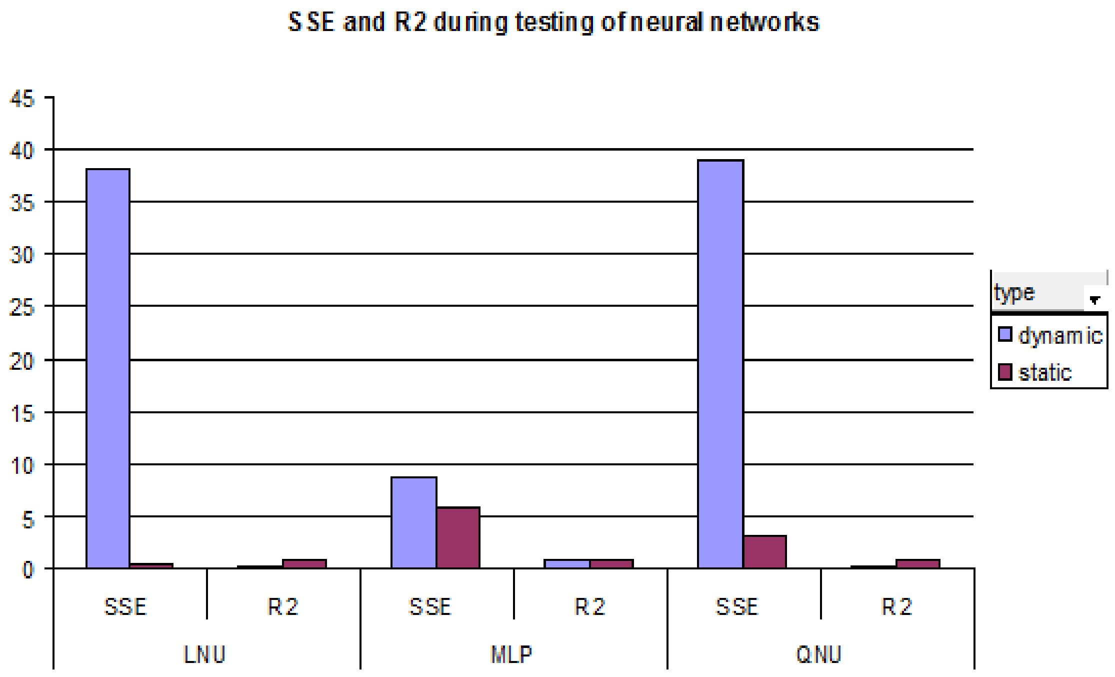

Fig.27 SSE and R^2 during testing

Conclusion

After completion of all required sections in this thesis, a number of key points with respect to its main objectives can be outlined. The primary goal of this thesis was the practical comparison of the different neural networks for approximation of the pneumatic muscle actuator. First of all, I reviewed recently published papers on studies of PMA and the ways to control it with a strong emphasis on the neural network approach. There are various methods for dealing with PMA, and, as one could notice, the neural networks are slowly gaining popularity in the field of PMA control due to their numerous advantages, such as performing less time consuming training, simplicity of implementation, ability to quickly detect complex nonlinear relationship between input and output data in case of the supervised training, and the possibility to choose the suitable training algorithms. The given model of PMA had the very specific issue concerning the behavior of the input-output system, which occurred to due the use of the pulse-width modulation, so that dealing with that issue was one of my main objectives as well. Next, I outlined the basic concepts of the most commonly used nowadays neural networks, namely LNU, QNU, and MLP, and the algorithms they could be trained with, because in order to achieve the desired results I had to choose ones that would be most suitable for this application. The next objective of this thesis was to carry out the experimental analysis on approximation of PMA where the attempt was made to find out which neural network shows better convergence and has a better ability to generalize. During the training phase I implemented Levenberg-Marquardt and a variation of Backpropagation Through Time algorithm to update weights and calculate new (trained) outputs. After the completion of training phase the testing procedure was performed in order to see if the neural networks were able to generalize the data. Then I also calculated the values of SSE and R^2, so it could help out with the evaluation of the best approximated neural network. However, the main issue of this work was handling the nonlinearity of the given model of PMA that occurred due to the presence of the pulse-width modulation (PWM), so that values of SSE and R^2 weren't the solid factors I could completely rely on. For instance the Static LNU had excellent values of SSE and R^2, but this neural network couldn't deal with the nonlinearity of the model (Fig.14). As in case of Dynamic LNU, QNU, and MLP (Fig.15, Fig.17, Fig.19), these neural networks suffered from poor capability for training and generalization due to the same problem caused by PWM, however dynamic MLP indicated better capability to handle approximate PMA than QNU in this work. As a conclusion I could say that the static MLP (Fig.18, Fig.24) showed the best performance during both training and testing compared to other tested neural networks, and the values of SSE and R^2 also corresponds to this assumption. Importantly, the static MLP was best in dealing with the specific behavior of this specific SISO system, while others couldn't seem to handle this issue (two-valve PWM actuated PMA). With this result I can see more research should be provided in the field of approximation by means of the neural network approach: larger sets of data can be used and other techniques can be implemented in order to apprehend the nonlinear behavior of the PWM actuated PMA. Practically, MLP networks resulted as the best candidate out of LNU and QNU for approximation of PWM actuated PMA in this case study and can be considered in a control design.

References

[1] Prashant, J., S. Q. Xie, Shahid, H., Kean, A.: "Modeling pneumatic muscle actuators: artificial intelligence approach", *International Journal of Information Acquisition Vol. 7, No. 2- 151,* 2010.

[2] Tomoyuki M., Nobutaka T., Takayuki K., Yoichiro, N., Mitsumasa, S.: "Spring-damper model and articulation control of pneumatic artificial muscle actuators", *Robotics and Biomimetics (ROBIO), 2011 IEEE International Conference*, 7-11 Dec. 2011.

[3] Piteľ, J.: "Modelling of the PAM Based Antagonistic Actuator*", Cybernetic letters ISSN 1802-3525,* April 2008.

[4] Borziková, J., Pitel, J., Tóthová, M., Sulc, B.: "Dynamic simulation model of PAM based antagonistic actuator", *Carpathian Control Conference (ICCC), 2011 12th International*, 25-28 May 2011.

[5] AGARD lecture series, "Artificial Neural Network Approaches in Guidance and Control", *40 Chigwell Lane, Loughton, Essex,* 1991.

[6] Prashant, J., Xie, S.: "Artificial Neural Network based dynamic modelling of indigenous pneumatic muscle actuators", *Mechatronics and Embedded Systems and Applications (MESA), 2012 IEEE/ASME International Conference*, 8-10 July 2012.

[7] Boudoua, S., Hamerlain, F., Hamerlain, M.: "Neuro sliding mode based chatter free control for an artificial muscles robot arm", *Neural Networks (IJCNN), The 2010 International Joint Conference*, 18-23 July 2010.

[8] Hošovský, A., Havran, M.: "Dynamic modelling of one degree of freedom pneumatic muscle-based actuator for industrial applications", *Tehnički vjesnik,* Vol.19 No.3 Rujan, 2012.

[9] Hošovský, A., Novák-Marcinčin, J., Piteľ, J., Boržíková, J., Židek, K.: "Model-based Evolution of a Fast Hybrid Fuzzy Adaptive Controller for a Pneumatic Muscle Actuator", *International Journal of Advanced Robotic Systems. ,p.1-11. - ISSN 1729-8806,* Vol. 9 (56)- 2012

[10] Gupta, M., M., Bukovsky, I., Homma, N. , Solo M. G. A., Hou Z.-G.: "Fundamentals of Higher Order Neural Networks for Modeling and Simulation", in *Artificial Higher Order Neural Networks for Modeling and Simulation*, ed. M. Zhang, IGI Global, 2012.

[11] Bukovsky, I., Kei, I., Noriyasu, H., Yoshizawa, M., Rodriguez, R.: "Testing Potentials of Dynamic Quadratic Neural Unit for Prediction of Lung Motion during Respiration for Tracking Radiation Therapy", *Neural Networks (IJCNN), The 2010 International Joint Conference*, 18-23 July 2010

[12] Khotanzad, A., Chung, C.: "Application of multi-layer perceptron neural networks to vision problems", *Neural Computing & Applications,* Volume 7, Issue 3, pp 249-259, 1998.

[13] Rodriguez, R.: "Lung Tumor Motion Prediction by Neural Networks", (Supervisor Ivo Bukovsky) *Student's Conference STC,* Faculty of Mechanical Engineering, CTU in Prague November 2012

[14] Bukovsky, I., Homma, N., Smetana, L., Rodriguez, R., Mironovova M., Vrana S.: "Quadratic Neural Unit is a Good Compromise between Linear Models and Neural Networks for Industrial Applications", *ICCI 2010 The 9th IEEE International Conference on Cognitive Informatics,* Tsinghua University, Beijing, China, July 7-9, 2010.

[15] Haykin, S.: *Neural Network: a Comprehensive Foundation,* second ed., Prentice Hall, Englewood, Cliffs, NJ, 2001.

[16] Ham, F.M., Kostanic, I.: Principles of Neurocomputing for Science and Engineering, McGraw-Hill, New York, 2001.

[17] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Comput*., vol. 1, pp. 270–280, 1989.

[18] P. J.Werbos, "Backpropagation through time: What it is and how to do it," *Proc. IEEE,* vol. 78, no. 10, pp. 1550–1560, Oct. 1990.

[19] Jaeger, H.: A tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach, 3rd revision: April 2008 (online 5/2013, minds.jacobs-university.de/.../ESNTutorialRev.pdf)

[20] Benes, M.: "Software Application for Adaptive Identification and Controller Tuning", (Supervisor Ivo Bukovsky) *Student's Conference STC,* Faculty of Mechanical Engineering, CTU in Prague 2013

[21] Dodge, Y.: "The Concise Encyclopedia of Statistics", Springer book archives, 2008

[22] Nagelkerke, N. J. D.:"A Note on a General Definition of the Coefficient of Determination". Biometrika 78 (3), 1991.

[23] Chiang, M.Y., Li-Chiu Chang: “Comparison of static-feedforward and dynamic-feedbackneural networks for rainfall–runoff modeling”, *Journal of Hydrology 290 297–311*, November, 2003.

[24] Veselíny, M., Líška, O., Bukovský, I., Jobbágy, B.: “Viacnásobné využitie umelej inteligencie v modernom rehabilitačnom zariadení”, Automatizácia a riadenie v teórii a praxi ARTEP 2013 workshop odborníkov z univerzít, vysokých škôl a praxe 20. 2. – 22. 2. 2013 Stará Lesná, SR.

Appendix

Static LNU:

```
#importing modules
from numpy import *
from matplotlib.pyplot import *
from numpy.random import randn, rand
from numpy.linalg import inv

#----initializing files with real data
##Selecting only first Ntrain samples of TRAINING DATA

Ntrain=500

u=loadtxt('u.txt')
u=u[0:500]
meanu=mean(u)
stdu=std(u)
yr=loadtxt('y.txt')
yr=yr[0:500]
stdy=std(yr)
meany=mean(yr)
N=len(u)

# resampling
dn=1     # 1... no resampling
if dn>1:
    u=u[range(0,N,dn)]
    yr=yr[range(0,N,dn)]
N=len(u)

#standardization of data
yr=(yr-meany)/3/stdy
u=(u-meanu)/3/stdu

#setup
nyr=12 #range of y in the model system
nur=12 #range of u in the model system
nx=1+nyr+nur
nw=nx
mu=1#learning rate
epochs=100
w=randn(nw)/nw #keeping weights small
x=ones(nx) #x[0]=1


J=ones((N,nw)) #Jacobian matrix with N x nw dimension
L=eye(nw)
SSE=zeros(epochs)
y=zeros((N))
e=zeros((N))
Nstart=max(nyr,nur)
y[0:Nstart+1]=yr[0:Nstart+1] # i.c.
```

```
for epoch in range(epochs):
    for k in range(Nstart,N-1):
        x[range(1,nyr,1)]=yr[range(k,k-nyr,-1)]
        x[range(nyr+1,nx,1)]=u[range(k,k-nur,-1)]
        J[k+1:]=x
    y=dot(w, J.T) # does no calculate values up to Nstart
    y[0:Nstart+1]=yr[0:Nstart+1]
    e=yr-y

    JJ=J[Nstart:,:]
    ee=e[Nstart:]
    dw=dot(dot(inv(dot(JJ.T,JJ)+1/mu*L), JJ.T),ee)

    w=w+dw
    SSE[epoch]=dot(ee,ee)
    print(SSE[epoch])
# de-normalization
u=u*3*stdu+meanu
y=y*3*stdy+meany
yr=yr*3*stdy+meany

figure()
subplot(221)
plot(u,color="black", linewidth=1.5, linestyle="-")
plot(yr,color="blue", linewidth=1.5, linestyle="-")
legend(loc='upper left')
plot(y,color="green", linewidth=1.5, linestyle="-")
subplot(222)
plot(u,'k')
plot(yr,color="blue", linewidth=2.5, linestyle="-")
legend(loc='upper left')
plot(y,color="green", linewidth=2.5, linestyle="-")
subplot(223)
semilogy(SSE,color="red", linewidth=2.5, linestyle="-")
subplot(224)
plot(w,color="black", linewidth=1.5, linestyle="-")


#----TESTING of the trained neural architecture
# on the rest of data
u=loadtxt('u.txt')
u=u[Ntrain-1:]
yr=loadtxt('y.txt')
yr=yr[Ntrain-1:]
N=len(u)


# resampling
##dn=1     # must be same as in training
if dn>1:
    u=u[range(0,N,dn)]
    yr=yr[range(0,N,dn)]
N=len(u)
```

```
# resampling
##dn=1      # must be same as in training
if dn>1:
    u=u[range(0,N,dn)]
    yr=yr[range(0,N,dn)]
N=len(u)

#standardization of data
yr=(yr-meany)/3/stdy
u=(u-meanu)/3/stdu
y=yr.copy() # covers also i.c.


x=ones(nx)  #x[0]=1
for k in range(Nstart,N-1):
    x[range(1,nyr,1)]=yr[range(k,k-nyr,-1)]
    x[range(nyr+1,nx,1)]=u[range(k,k-nur,-1)]
    y[k+1]=dot(w, x)  # does no calculate values up to Nstart
    e=yr-y

figure()
plot(u,color="black", linewidth=2.5, linestyle="-"),legend('u')
plot(yr,color="blue", linewidth=2.5, linestyle="-"), xlabel('k')
legend(loc='upper left')
```

Dynamic LNU: Same initial setup as in case of Static LNU, training is performed as follows:

```
J=ones((N,nw)) #Jacobian matrix with N x nw dimension
L=eye(nw)
SSE=zeros(epochs)
y=zeros((N))
y[0:nx]=yr[0:nx]    # i.c. for dynamical LNU
e=zeros((N))

Nstart=max(nyr,nur)
y[0:Nstart+1]=yr[0:Nstart+1] # i.c.

for epoch in range(epochs):
    for k in range(Nstart,N-1):
        x[range(1,nyr,1)]=y[range(k,k-nyr,-1)]
        x[range(nyr+1,nx,1)]=u[range(k,k-nur,-1)]
        J[k+1:]=x
        y[k+1]=dot(w,x)
    e=yr-y

    JJ=J[Nstart:,:]
    ee=e[Nstart:]

    dw=dot(dot(inv(dot(JJ.T,JJ)+1/mu*L), JJ.T),ee)
    w=w+dw
    SSE[epoch]=dot(ee,ee)
    print(SSE[epoch])
# de-normalization
u=u*3*stdu+meanu
y=y*3*stdy+meany
yr=yr*3*stdy+meany
```

Testing of Dynamic LNU:

```
##===================================
#----TESTING of the trained neural architecture
# on continuing data
u=loadtxt('u.txt')
u=u[Ntrain-1:]
yr=loadtxt('y.txt')
yr=yr[Ntrain-1:]
N=len(u)
# resampling
##dn=1     # must be same as in training
if dn>1:
    u=u[range(0,N,dn)]
    yr=yr[range(0,N,dn)]
N=len(u)
#standardization of data
yr=(yr-meany)/3/stdy
u=(u-meanu)/3/stdu
y=yr.copy() # covers also i.c.
x=ones(nx)  #x[0]=1
for k in range(Nstart,N-1):
    x[range(1,nyr,1)]=y[range(k,k-nyr,-1)]
    x[range(nyr+1,nx,1)]=u[range(k,k-nur,-1)]
    y[k+1]=dot(w, x) # does no calculate values up to Nstart
    e=yr-y
```

Static QNU:

```
#setup
nyr=6
nur=6
nx=1+nyr+nur
nw=(nx**2+nx)/2

mu=0.1
epochs=200
w=randn(nw)/nw
y=zeros((N))
y[0:nx]=yr[0:nx]
e=zeros((N))
L=eye(nw)
SSE=zeros(epochs)
x=ones(nx)
colx=ones(nw)


Nstart=max(nyr,nur)
y[0:Nstart+1]=yr[0:Nstart+1] # i.c.

J=ones((N,nw))
for epoch in range(epochs):
    for k in range(Nstart,N-1):
        x[range(1,nyr,1)]=yr[range(k,k-nyr,-1)]
        x[range(nyr+1,nx,1)]=u[range(k,k-nur,-1)]
        pom=0
        for i in range (0,nx):
            for j in range (i,nx):
                colx[pom]=x[i]*x[j]
                pom=pom+1
        J[k+1,:]=colx
        y=dot(w, J.T)
        y[0:Nstart+1]=yr[0:Nstart+1]
        e=yr-y

        JJ=J[Nstart:,:]
        ee=e[Nstart:]

        dw=dot(dot(inv(dot(JJ.T,JJ)+1/mu*L), JJ.T),ee)

        w=w+dw
        SSE[epoch]=dot(ee,ee)
        print SSE[epoch]


    # de-normalization
    u=u*3*stdu+meanu
    y=y*3*stdy+meany
    yr=yr*3*stdy+meany
```

Testing of Static QNU:

```
    sigma1=zeros((N))
    sigma2=zeros((N))
    x=ones(nx) #x[0]=1
    colx=ones(nw)
    J=ones((N,nw))
    for k in range(Nstart,N-1):
        x[range(1,nyr,1)]=yr[range(k,k-nyr,-1)]
        x[range(nyr+1,nx,1)]=u[range(k,k-nur,-1)]
        pom=0
        for i in range (0,nx):
            for j in range (i,nx):
                colx[pom]=x[i]*x[j]
                pom=pom+1
        J[k+1,:]=colx
        y=dot(w, J.T)
        y[0:Nstart+1]=yr[0:Nstart+1]
        e=yr-y
        JJ=J[Nstart:,:]
        ee=e[Nstart:]
        SSE=dot(ee,ee)
        print (SSE)

    # de-normalization
    u=u*3*stdu+meanu
    y=y*3*stdy+meany
    yr=yr*3*stdy+meany
```

Dynamic QNU:

```
mu=0.0001
epochs=200
w=randn(nw)/nw
y=zeros((N))
y[0:nx]=yr[0:nx]
e=zeros((N))
L=eye(nw)
SSE=zeros(epochs)
x=ones(nx)
colx=ones(nw)
Nstart=max(nyr,nur)
y[0:Nstart+1]=yr[0:Nstart+1] # i.c.

J=ones((N,nw))
for epoch in range(epochs):
    for k in range(Nstart,N-1):
        x[range(1,nyr,1)]=y[range(k,k-nyr,-1)]
        x[range(nyr+1,nx,1)]=u[range(k,k-nur,-1)]
        pom=0
        for i in range (0,nx):
            for j in range (i,nx):
                colx[pom]=x[i]*x[j]
                pom=pom+1
        J[k+1,:]=colx
        y[k+1]=dot(w, colx)
        e[k+1]=yr[k+1]-y[k+1]

    JJ=J[Nstart:,:]
    ee=e[Nstart:]

    dw=dot(dot(inv(dot(JJ.T,JJ)+1/mu*L), JJ.T),ee)

    w=w+dw
    SSE[epoch]=dot(ee,ee)
    print SSE[epoch]
```

Testing of dynamic QNU:

```
for k in range(Nstart,N-1):
    x[range(1,nyr,1)]=y[range(k,k-nyr,-1)]
    x[range(nyr+1,nx,1)]=u[range(k,k-nur,-1)]
    pom=0
    for i in range (0,nx):
        for j in range (i,nx):
            colx[pom]=x[i]*x[j]
            pom=pom+1
    J[k+1,:]=colx
    y=dot(w, J.T)
    y[0:Nstart+1]=yr[0:Nstart+1]
    e=yr-y
    JJ=J[Nstart:,:]
    ee=e[Nstart:]

    SSE=dot(ee,ee)
    print SSE
```

Static MLP:

```
from numpy import *
from matplotlib.pyplot import *
from numpy.random import randn, rand
from numpy.linalg import inv

#---defining functions
def phi(ny):
    bz=2./(1.+exp(-ny))-1
    return(bz)
def dphidny(ny):
    bzz=2*exp(-ny)/(1.+exp(-ny))**2
    return(bzz)

#----initializing files with real data
##Selecting only first Ntrain samples of TRAINING DATA

Ntrain=500

u=loadtxt('u.txt')
u=u[0:500]
meanu=mean(u)
stdu=std(u)
yr=loadtxt('y.txt')
yr=yr[0:500]
stdy=std(yr)
meany=mean(yr)
N=len(u)

# resampling
dn=1     # 1... no resampling
if dn>1:
    u=u[range(0,N,dn)]
    yr=yr[range(0,N,dn)]
N=len(u)


#setup

nyr=5 #range of y in dynamic model system
nur=10 #range of u in dynamic model system
muW=0.1 #learning rate for hidden layer
muV=0.05 #learning rate for output neurons
n1=5 #neurons in hidden layer
nx=1+9 #length of the input vector
nv=1+n1   #number of output neuron weights
nxi=nv #number of xi
epochs=300

#---INITIALIZATION---

#weights
W=randn(n1,nx)/nx   #keeping init weights  small
v=randn(nv)/nv
e=zeros(N)
y=zeros(N) #network output
```

```
y[0:nyr]=yr[0:nyr]
xi=ones(nxi)
dxidny=zeros(n1+1)
dydv=zeros((N,nv))
Lv=eye(nv)
Lw=eye(nx)
dydW=zeros((N,nx,n1))
SSE=zeros(epochs)
x=ones(nx)

Nstart=max(nyr,nur)
y[0:Nstart+1]=yr[0:Nstart+1] # i.c.

for epoch in range(epochs):
    for k in range(Nstart,N-1):
        x[range(1,nyr,1)]=yr[range(k,k-nyr,-1)]
        x[range(nyr+1,nx,1)]=u[range(k,k-nur,-1)]
        #x[2]=u[k]
        ny=dot(W,x)   #
        #phi=2/(1+exp(-ny))-1
        xi[1:]=phi(ny)
        y[k]=dot(v,xi)
        e[k]=yr[k]-y[k]
        #derivatives for jacobians
        #output neuron
        dydv[k,:]=xi
        #hidden neurons
        dxidny[1:]=dphidny(ny)
        #for  neuron in range(n1):
        for i in range(1,n1+1): #for all weights of each neuron
            dydW[k,:,i-1]=v[i]*dxidny[i]*x
    #update output neuron
    Jv=dydv
    dv=dot(dot(inv(dot(Jv.T,Jv)+1./muV*Lv),Jv.T),e)
    v=v+dv
    #update hidden neuron
    for i in range(1,n1+1):
        Jw=dydW[:,:,i-1]
        dw=dot(dot(inv(dot(Jw.T,Jw)+1./muW*Lw),Jw.T),e)
        W[i-1,:]=W[i-1,:]+dw
    ee=e[Nstart:]
    SSE[epoch]=dot(ee,ee)
    print SSE[epoch]

# de-normalization
u=u*3*stdu+meanu
y=y*3*stdy+meany
yr=yr*3*stdy+meany
```

Testing of static MLP:

```
for k in range(Nstart,N-1):
    x[range(1,nyr,1)]=yr[range(k,k-nyr,-1)]
    x[range(nyr+1,nx,1)]=u[range(k,k-nur,-1)]
    ny=dot(W,x)
    xi[1:]=phi(ny)
    y[k]=dot(v,xi)
    e=yr-y
    ee=e[Nstart:]
    SSE=dot(ee,ee)
    print SSE

for k in range (Nstart, N-1):
    sigma1=yr[k]-y[k]
    sigma2=yr[k]-meany

R=1-(sum(sigma1)/sum(sigma2))

print('R=', R)
```

Dynamic MLP:

```
#setup

nyr=5 #range of y in dynamic model system
nur=10 #range of u in dynamic model system
muW=0.1 #learning rate for hidden layer
muV=0.05 #learning rate for output neurons
n1=5 #neurons in hidden layer
nx=1+9 #length of the input vector
nv=1+n1   #number of output neuron weights
nxi=nv #number of xi
epochs=500

#---INITIALIZATION---

#weights
W=randn(n1,nx)/nx   #keeping init weights  small
v=randn(nv)/nv
e=zeros(N)
y=zeros(N) #network output
y[0:nyr]=yr[0:nyr]
xi=ones(nxi)
dxidny=zeros(n1+1)
dydv=zeros((N,nv))
Lv=eye(nv)
Lw=eye(nx)
dydW=zeros((N,nx,n1))
SSE=zeros(epochs)
x=ones(nx)

Nstart=max(nyr,nur)
y[0:Nstart+1]=yr[0:Nstart+1] # i.c.
```

```
 for epoch in range(epochs):
     for k in range(Nstart,N-1):
         x[range(1,nyr,1)]=y[range(k,k-nyr,-1)]
         x[range(nyr+1,nx,1)]=u[range(k,k-nur,-1)]
         #x[2]=u[k]
         ny=dot(W,x)   #
         #phi=2/(1+exp(-ny))-1
         xi[1:]=phi(ny)
         y[k]=dot(v,xi)
         e[k]=yr[k]-y[k]
         #derivatives for jacobians
         #output neuron
         dydv[k,:]=xi
         #hidden neurons
         dxidny[1:]=dphidny(ny)
         #for  neuron in range(n1):
         for i in range(1,n1+1): #for all weights of each neuron
             dydW[k,:,i-1]=v[i]*dxidny[i]*x
     #update output neuron
     Jv=dydv
     dv=dot(dot(inv(dot(Jv.T,Jv)+1./muV*Lv),Jv.T),e)
     v=v+dv
     #update hidden neuron
     for i in range(1,n1+1):
         Jw=dydW[:,:,i-1]
         dw=dot(dot(inv(dot(Jw.T,Jw)+1./muW*Lw),Jw.T),e)
         W[i-1,:]=W[i-1,:]+dw

     ee=e[Nstart:]
     SSE[epoch]=dot(ee,ee)

     SSE[epoch]=dot(ee,ee)
     print SSE[epoch]
```

Testing of dynamic MLP:

```
##==================================
#----TESTING of the trained neural architecture


u=loadtxt('u.txt')
u=u[Ntrain-1:]
yr=loadtxt('y.txt')
yr=yr[Ntrain-1:]
N=len(u)


# resampling
##dn=1      # must be same as in training
if dn>1:
    u=u[range(0,N,dn)]
    yr=yr[range(0,N,dn)]
N=len(u)
sigma1=zeros((N))
sigma2=zeros((N))
#standardization of data
yr=(yr-meany)/3/stdy
u=(u-meanu)/3/stdu
y=yr.copy() # covers also i.c.
```

```
for k in range(Nstart,N-1):
    x[range(1,nyr,1)]=y[range(k,k-nyr,-1)]
    x[range(nyr+1,nx,1)]=u[range(k,k-nur,-1)]
    ny=dot(W,x)
    xi[1:]=phi(ny)
    y[k]=dot(v,xi)
    e=yr-y
    ee=e[Nstart:]
    SSE=dot(ee,ee)
    print SSE

for k in range (Nstart, N-1):
    sigma1=yr[k]-y[k]
    sigma2=yr[k]-meany

R=1-(sum(sigma1)/sum(sigma2))
```