Chapter 4

# Dual Heuristic Neural Programming Controller for Synchronous Generator

Mato Miskovic, Ivan Miskovic and Marija Mirosevic

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/58377

## 1. Introduction

This chapter describes the development of voltage control system of a synchronous generator based on neural networks. Recurrent (dynamic) neural networks (RNN) are used, as a type that has great capabilities in approximation of dynamic systems [1]. Two algorithms are used for training – Dual Heuristic Programming (DHP) and Globalized Dual Heuristic Programming (GDHP). The algorithms have been developed for the optimal control of nonlinear systems using dynamic programming principles.

Neural voltage controller is developed in MATLAB and Simulink environment. For training purposes a mathematical model of synchronous generator is designed and applied in Simulink. DHP and GDHP algorithms are designed in Simulink, with matrix calculations in S-functions. Algorithms are used for offline training of neural networks (NN). In the second part, the same functions are redesigned as real time controllers, based on the Real Time Windows Target Toolbox.

DHP or GDHP algorithms provide a significant improvement over conventional PI controller with, in some cases, power system stabilizer (PSS). Conventional linear controller can only provide optimal control in single operating point. Algorithms described in the chapter provide significantly better control over the whole operating range by minimization of the user defined cost function during the training of the neural network.

Digital voltage controller is implemented on real system in two basic steps. First step is neural network design and training in Matlab environment on desktop computer. Second step consists of transfer of controller with computed coefficients to the digital control system hardware.

For the controller to have optimal performance in all real operating conditions, neural networks must be trained for the whole working range of the plant. Procedure described in the chapter can be applied on standard voltage control systems by replacing the PI controller with the offline-trained neural network. Most modern digital control systems have sufficient hardware and software support for neural network implementation.

## 2. Recurrent neural networks

Neural networks are computational models capable of machine learning and pattern recognition. They can also be used as universal approximators of continuous real functions. Typical dynamic neural network is shown in Figure 1.

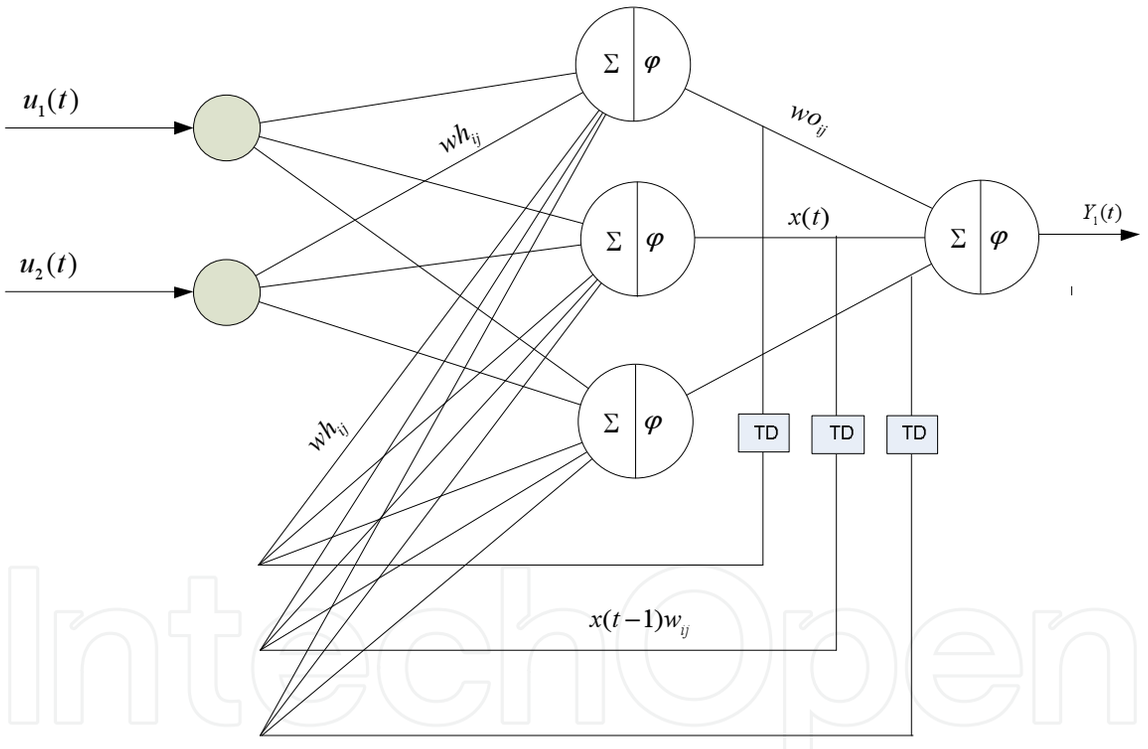



**Figure 1.** Recurrent neural network structure

Neural network is formed from interconnected layers of neurons. Output of each neuron is calculated from:

$$y_k = \varphi\left(\sum_{j=0}^{m}\left(w_{kj}x_{kj}\right)+b\right) \tag{1}$$

where

$\varphi$ is activation function,

$w_{kj}$ and $x_{kj}$ are weights and outputs from previous layer and $b$ is bias.

Typical activation function maps input values $u(t) \in [-\infty, \infty]$ to output $y(t) \in [0, 1]$ or $y(t) \in [-1, 1]$. Activation function must be derivable over the whole range. Among typically used nonlinear activation functions are *logsig*, *tansig* and Gaussian.

Dynamic behavior is added by introducing a memory element (step delay) that stores neuron output and reintroduces it as input to all neurons in the same layer in next time step.

This type of network is named recurrent or dynamic neural network. An example of tansig (*Hyperbolic Tangent Sigmoid Transfer Function*) activation function, $\phi(u) = \frac{2}{1+e^{-2u}} - 1$, and its derivative $\phi(u)' = 1 - \phi(u)^2$ is shown in Figure 2.

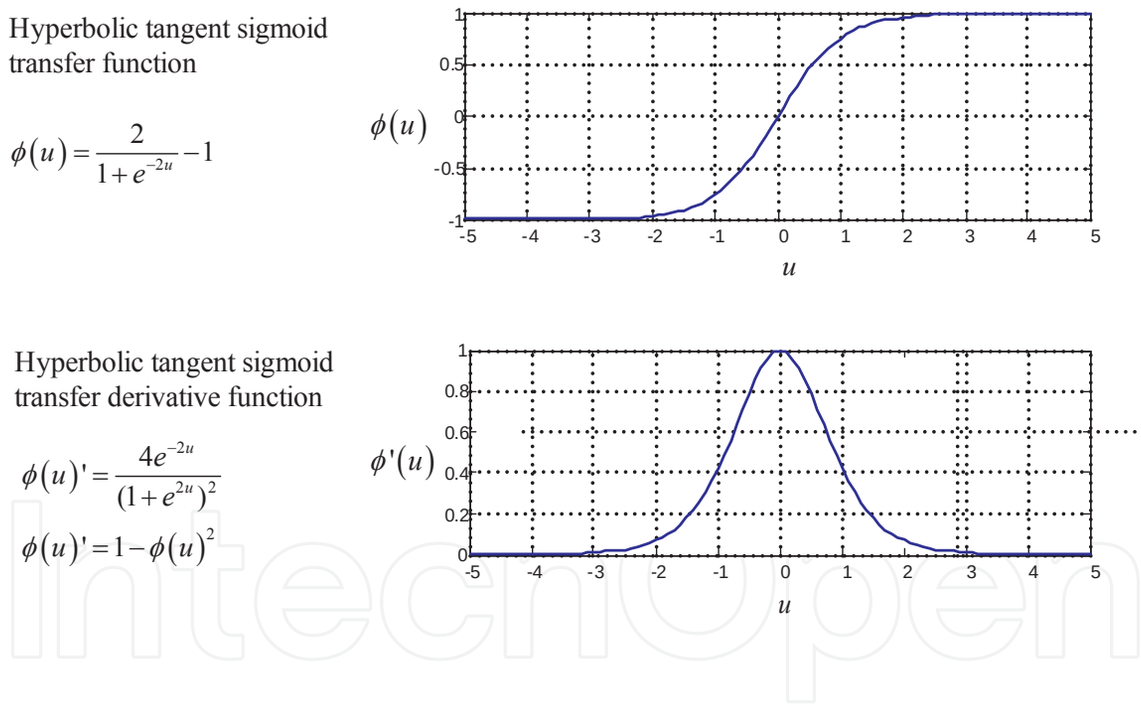



**Figure 2.** Tansig activation function and its derivative

Input values are in range of $u \in [-\infty, \infty]$, with output values in range $\varphi(u) \in [-1, 1]$.

Use of function with simple derivative significantly improves calculation times of numerical procedures.

Dynamic (recurrent) neural networks provide good approximation of time-dependent functions. However, training of recurrent neural networks is more complex than training static networks.

### 2.1. Recurrent neural networks training

There are many methods for training neural networks, and most of them rely on some form of backpropagation – calculation of partial derivative of system output over network weight coefficients $\left(\frac{\partial y}{\partial w}\right)$. Partial derivatives are determined using chain rule, by finding derivatives of weight coefficients in output layer, using intermediate results to calculate next layer, with respect to network structure. Calculation is repeated until all layers are processed.

Described procedure is valid for static neural networks, but is of no use in recurrent networks, as each neuron has inputs from previous time step from the same layer. Two procedures are often used in calculation of partial derivatives: *Backpropagation Through Time* (*BPTT*) and *Real Time Recurrent Learning* (*RTRL*).

In BPTT recurrent neuron inputs are replaced with the same neural network delayed by one time step. Process is iterated N times. Thus, recurrent network is approximated by N feedforward networks with delayed outputs. Network weight coefficients, recurrent inputs and outputs must be preserved for past N time steps. Obtained derivative $\left(\frac{\partial y(t)}{\partial w(t)}\right)$ is used to determine new weight coefficients. One advantage of BPTT is that all methods used for training feedforward networks can be utilized in training recurrent networks.

This chapter will describe use of RTRL procedure in training dynamic networks. Output of recurrent network with one hidden and one output layer has the following form:

$$\begin{aligned} x(t) &= \tanh\left(W_1 \cdot p_1\right) \\ y(t) &= W_2 \cdot x(t) \end{aligned} \tag{2}$$

where

$p_1$– hidden layer input, $p_1=[u(t)\,;\,x(t-1)\,;\,1]$,

$x(t)$– hidden layer output,

y(t) – network output,

$W_1$– hidden layer weight coefficients,

$W_2$– output layer weight coefficients, $p_2=[x(t)\,;\,1]$

Equation (2) can easily be expanded for multi layer recurrent networks. Partial derivative of network output over output layer weight coefficients is

$$\frac{\partial y(t)}{\partial w_{ij}^{o}(t)} = p_2(t) \tag{3}$$

For hidden layer, partial derivatives are

$$\frac{\partial y(t)}{\partial w_h(t)} = \frac{\partial y(t)}{\partial w_2(t)} \frac{\partial x(t)}{\partial w_2(t)} \tag{4}$$

Partial derivative of hidden layer output over weight coefficients is

$$\frac{\partial x_l(t)}{\partial w_{i,j}(t)} = f\left[x(t)\right]'_i \left( p_j(t)\delta_{li} + \sum_{k=1}^{N_{W1}} w(t)_{l,N_{in}+k} \frac{\partial x_k(t-1)}{\partial w_{i,j}(t-1)} \right) \tag{5}$$

where

$N_{W1}$– number of neurons in hidden layer,

$N_{in}$– number of neurons in input layer.

Value of $\frac{\partial x_l(t)}{\partial w_{i,j}}$ is calculated using four nested loops (Appendix A).

Matrix of output derivatives over weight coefficients is of the form $[N_{W1},\ N_{W1} \cdot (N_{in} + N_{W1} + 1)]$.

### 2.2. Neural network training algorithms

#### *2.2.1. Gradient descent*

Gradient descent is the most commonly used method in neural networks training. Difference between real and desired network output can be expressed as

$$e(t) = y(t) - \hat{y}(t) \tag{6}$$

Differentiation of (5) provides update values of weight coefficients:

$$w_{ij}(t) = w_{ij}(t-1) - \beta \frac{\partial \hat{y}(t)}{\partial w_{ij}(t)} \tag{7}$$

Parameter $\beta$ defines learning rate and must be in range $0<\beta<1$.

*2.2.2. Kalman filter based training*

Kalman filter [5] was developed as a recursive procedure of optimal filtering of linear dynamic state-space systems. For RNN training extended Kalman filter (EKF) is used. It is also applicable to nonlinear systems using linearization around operating point. The algorithm is used as a predictive-corrective procedure.

Kalman filter based recurrent network training is described in [6] and [7].

Training is performed in two steps. The first step is calculation of prediction values of weight coefficients. Second step is correction of predicted variables based on measured state space.

Algorithm is defined by the following relations:

$$K(t) = P(t)H(t)\left(\eta(t)I + H^T(t)P(t)H(t)\right) \tag{8}$$

$$W(t) = W(t-1) + K(t)\varepsilon(t) \tag{9}$$

$$P(t+1) = P(t) - K(t)H^T(t)P(t) + Q(t) \tag{10}$$

where

$K(t)$– Kalman gain matrix, determined form network output derivatives over weight coefficients $H(t) = \frac{\partial y(t)}{\partial w}$. System linearization is accomplished by calculating $H(t)$.

$P(t)$– Error covariance matrix, calculated in every iteration

To escape local minima during training, a diagonal matrix $Q(t)$ is added to covariance matrix $P(t)$ in each time step.

Updates to weight coefficients are determined from (9), as a product of Kalman gain and $\varepsilon(t)$ in current step.

Training based on Kalman filter is fast, has good rate of convergence and is often the only practical solution for recurrent networks training. However, the procedure is demanding in terms of computational power and memory requirements.

## 3. System identification

Nonlinear dynamic systems are identified as input-output system model, or as a state space system. For the selected process, identification is performed by minimization of the cost function

$$E(t) = \frac{1}{2} \cdot \left[ y(t) - \hat{y}(t) \right]^T \cdot \left[ y(t) - \hat{y}(t) \right] \tag{11}$$

where $[y(t) - \hat{y}(t)]$ is the difference of the measured system outputs and output values of the model.

Selected cost function represents squared error of the identified model.

Dynamic system identification can be performed according to [9], using either series-parallel or parallel structure.

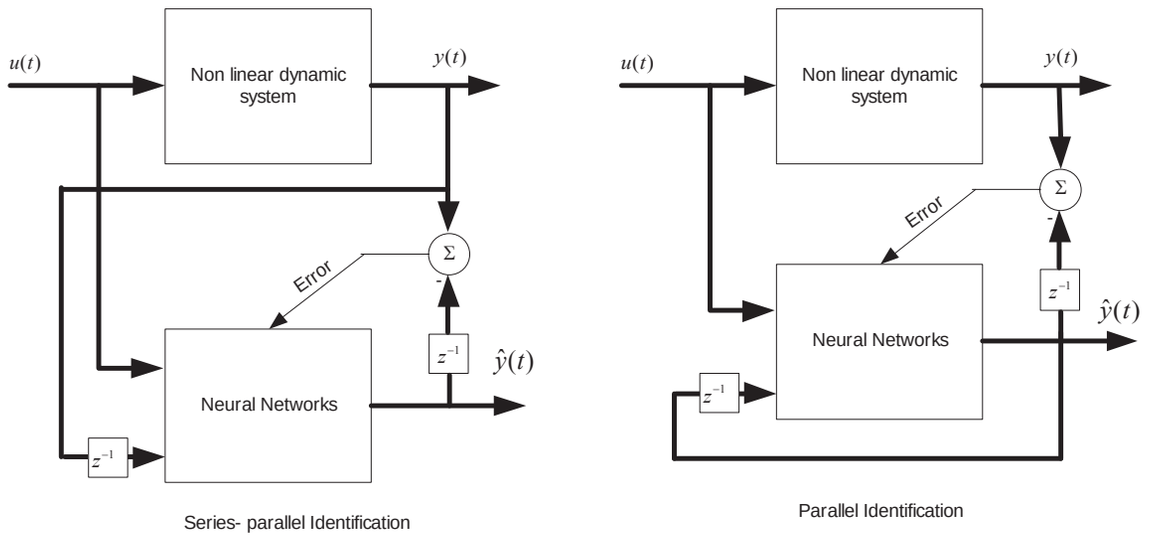



**Figure 3.** Nonlinear dynamic system identification

In series-parallel structure Figure 3, model neural network input vector is expanded with system outputs from previous steps. In parallel structure Figure 3, network output from previous states is added to input vector. Series-parallel identification provides estimated system output for next time step only, while parallel identification can predict multiple future iterations. However, parallel structure is harder to train and is susceptible to state drifting.

To verify the validity of the RTRL algorithm, neural networks were trained on several discrete mathematical functions. Supervised training has been used, with control value expressed as difference between desired and actual network output.

A parallel identification structure was used on dynamic SISO system $y(t+1) = \frac{y(t)}{1+y(t)^2} + u(t)^3$. Training has been performed on Recurrent Neural Network with four neurons in hidden layer, and one neuron in linear output layer. Identification results are shown in Figure 4.

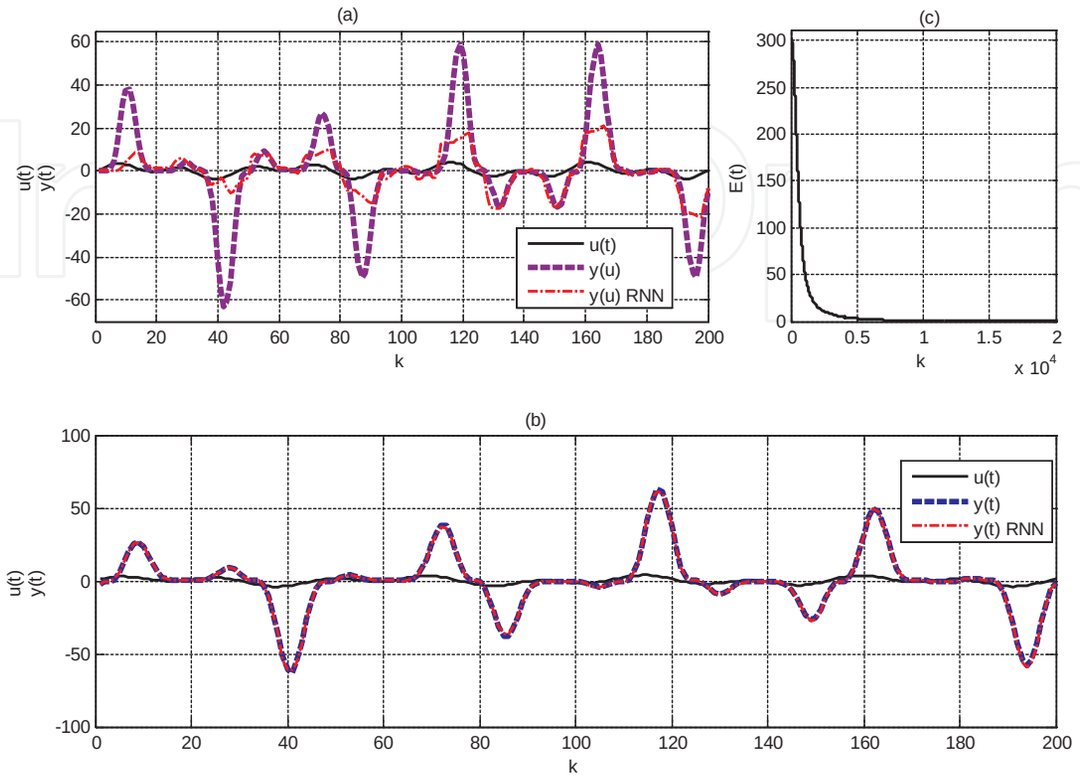



**Figure 4.** Identification of dynamic function 1

Figure 4a compares initial response of model output $y(t)$ and network output $\hat{y}(t)$. Figure 4b represents the same values with trained network. There is a perfect match between function output values and output of the trained NN. Figure 4c displays training rate as error square $(y(t)-\hat{y}(t))^2$. Training was completed in 20000 steps.

Input signal used for training and verification is $u(t)=\sin(\frac{2\pi}{50})+\sin(\frac{2\pi}{12})$, with sample time $(T_D=1s)$.

In second test, a system with one input, two state variables and one output has been identified:

$$x_1(t+1)=0.5x_2\cdot(t)+0.2\cdot x_1(t)\cdot x_2(t)$$
$$x_2(t+1)=-0.3\cdot x_1(t)+0.8\cdot x_2(t)+u(t)$$
$$y(t+1)=(x_1(t)+x_2(t))^2$$

System was identified using series-parallel structure with RNN with 5 neurons in hidden layer, and one linear neuron in output layer. Identification results are shown in Figure 5.

http://dx.doi.org/10.5772/58377

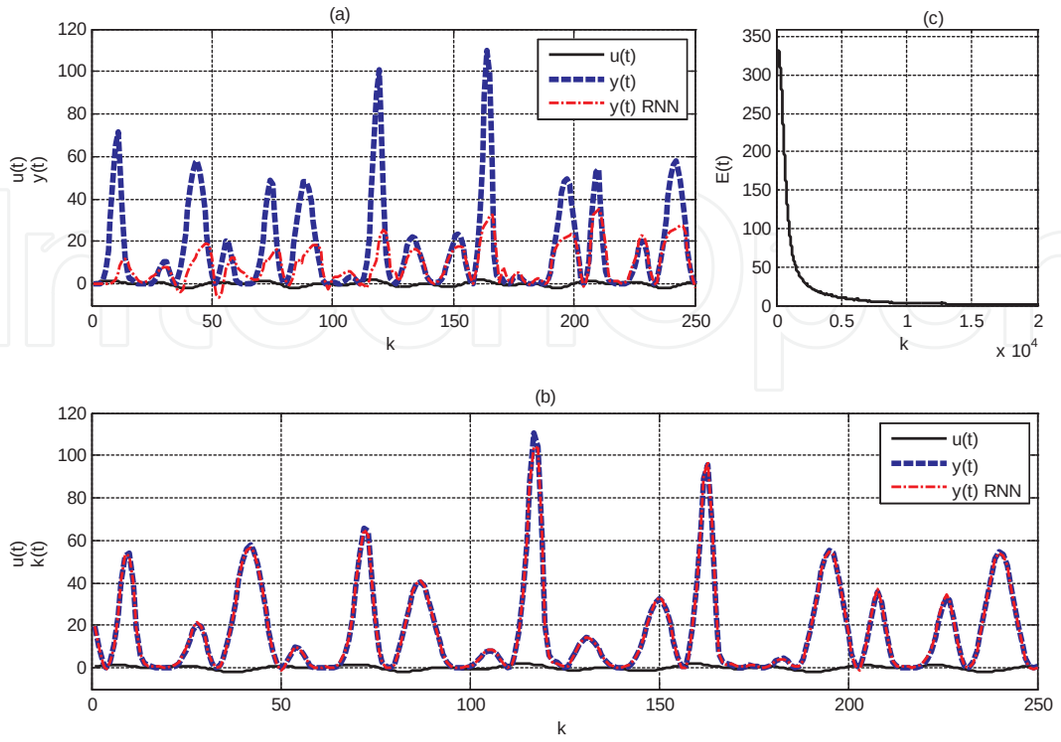



**Figure 5.** Identification of dynamic function 2

Input signal used for training and verification is the same as in previous example:

$$u(t)=\sin(\frac{2\pi}{50}kT_D)+\sin(\frac{2\pi}{22}kT_D),\ \text{ where } (T_D=1s).$$

Both examples were designed in Matlab and Simulink. Neural network was trained using c-language S-function (Appendix B). Although the training took 10000 steps, all important system dynamics were visible on trained network after 2000 steps.

### 3.1. Determination of partial derivatives of functions using neural networks

One of the most important features of neural network modeling is the possibility to determine system output derivatives over inputs $\left(\frac{\partial y_i(t)}{\partial u_j(t)}\right)$. After finding output derivatives, various optimization algorithms can be used.

Partial derivative is calculated from (2)

$$\frac{\partial y_i(t)}{\partial u_j(t)}=\sum_{j=1}^{N_{W1}}\left(W_{i,j}\cdot\sum_{k=1}^{N_p}\left(f_k'(x)\cdot W_{k,j}\right)\right) \tag{12}$$

where $N_p = N_{W1} + N_{in} + 1$ is number of hidden layer inputs. Partial derivative $\left(\frac{\partial y_i(t)}{\partial u_j(t)}\right)$ is obtained from network backpropagation.

For nonlinear function $y(t) = u(t-1)^2 + 6 \cdot u(t-1)^3$, partial derivative $\frac{\partial y(t)}{\partial u(t)}$ was first calculated analytically, and then obtained from model network using (12). Results are compared in Figure 6a and 6b.

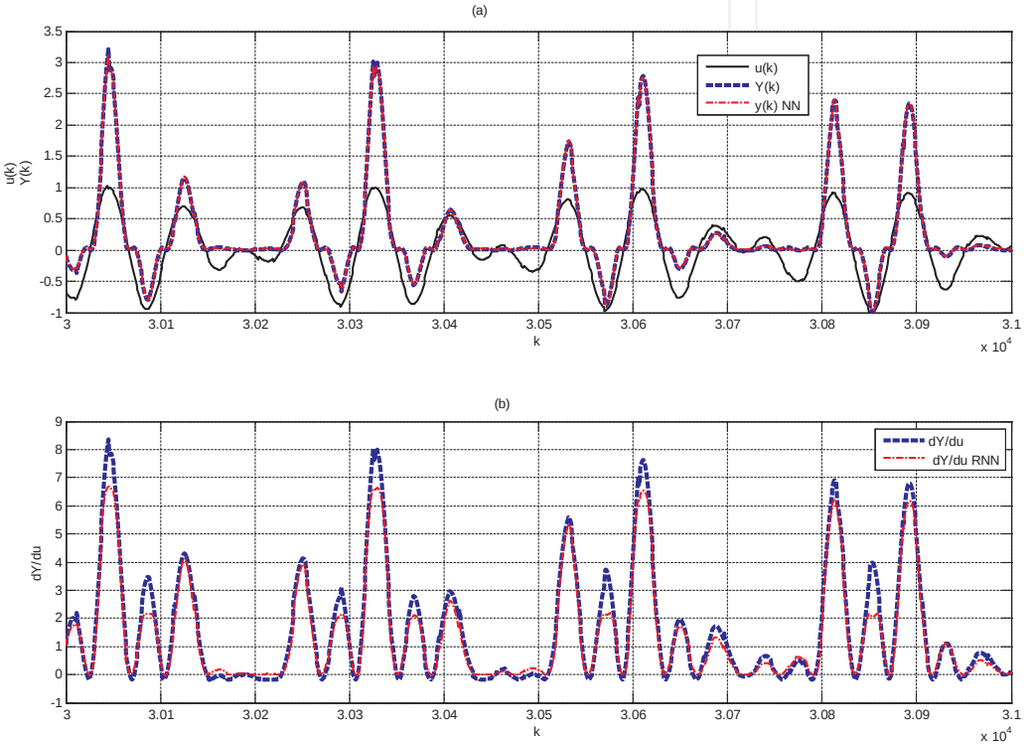



**Figure 6.** Identification of nonlinear function and calculation of partial derivatives using neural networks

Results from Figure 6b show good match between calculated function derivatives and values obtained from neural network of identified model.

In S-function for training RNN (Appendix B) a procedure is shown for online calculation of partial derivatives of $\left(\frac{\partial y_i(t)}{\partial u_j(t)}\right)$, using (12).

## 4. Adaptive critic design

Adaptive critic design (ACD) is a set of optimal control procedures of nonlinear systems. It is based on dynamic programming and neural networks. Optimal control is achieved by training

http://dx.doi.org/10.5772/58377

neural networks and using them in the structure according to the selected ACD algorithm. ACD algorithms are implemented using a heuristic approach – system state space variables, control values and cost function are selected based on process characteristics. This text focuses on the Heuristic Dynamic Programming (HDP), Dual Heuristic Programming (DHP) and General Dual Heuristic Programming (GDHP) [1,2].

### 4.1. Heuristic dynamic programming

Heuristic dynamic programming (*HDP*) is the simplest (by structure) ACD algorithm. The basis of all HDP algorithms is minimization of Bellman dynamic programming function $J(t)$:

$$J(t) = \sum_{k=0}^{\infty} \gamma^k U(t+k) \tag{13}$$

where

$\gamma$ represents discount factor with value in range $0 < \gamma < 1$, ensuring the convergence of Bellman sum $J(t)$ and cancellation of old values,

$U(t+1)$ is user defined utility function based on system state space variables.

Control signal that minimizes Bellman sum is generated using one of the Adaptive Critic algorithms. In practical control systems, HDP optimization is realized with three neural networks with a structure as shown in Figure 7.

Model neural network in *7a* is used for model identification. Network inputs are control signal $u(t)$ and output state in the previous time step $y(t-1)$. Network output represents estimated model output in the current time step $\hat{y}(t)$. Only systems outputs used in utility function need to be estimated. Model NN is structured according to the selected process model, where the control value is input to the NN, and NN output represents predicted system output. The same NN output values are used to calculate the cost function.

Model neural network is trained to minimize the quadratic criterion:

$$\|E_M\| = \sum_t E_M^T(t) \cdot E_M(t) \tag{14}$$

where:

$$E_M = y(t) - \hat{y}(t) \tag{15}$$

For adaptive control systems, model NN is trained continuously, allowing for adaptation to change of system parameters, and optimal control. Model neural network is trained simultaneously with other networks over time.

Critic neural network in *7b* is used for identification of Bellman sum $J(t)$.

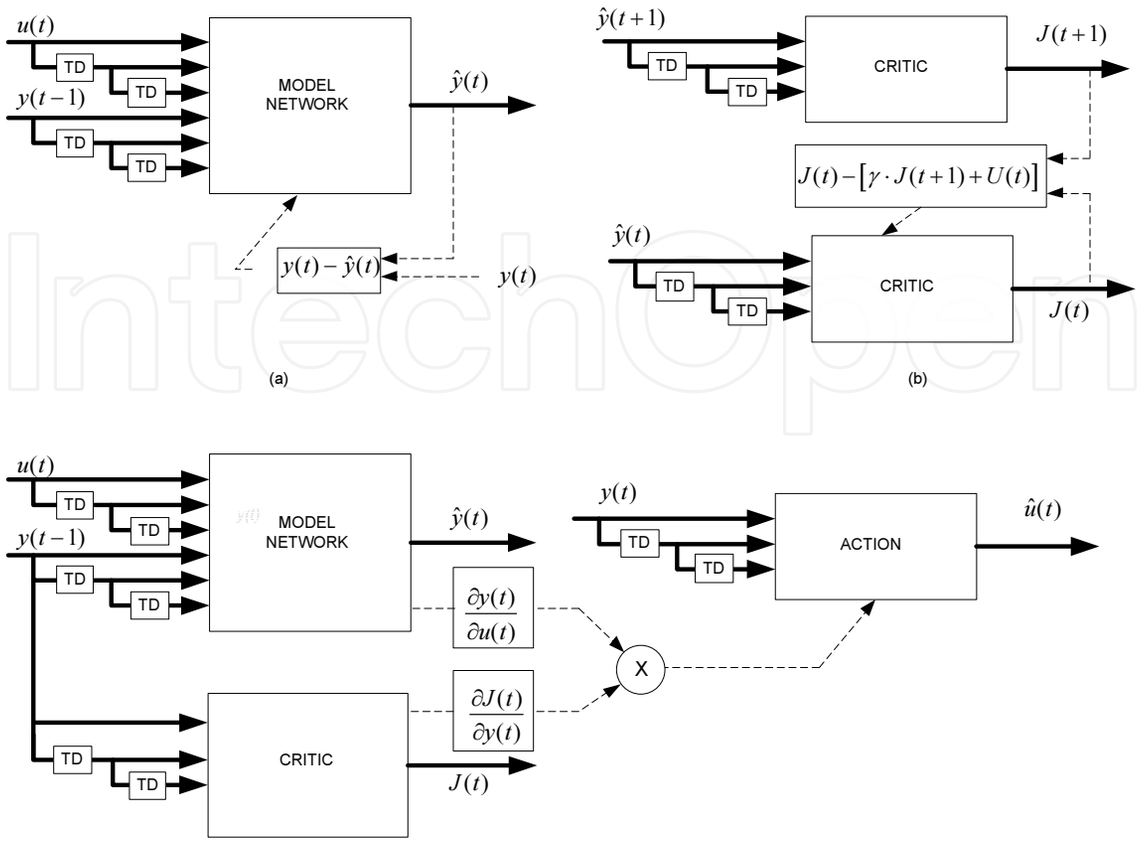



**Figure 7.** HDP Control structure

Training is used to minimize

$$\|E_C\| = \sum_t E_C^T(t) \cdot E_C(t) \tag{16}$$

where:

$$E_C(t) = J(t) - [\gamma \cdot J(t+1) + U(t)] \tag{17}$$

Critic neural network training is performed with a copy of the same network which is fed with predicted future system outputs $\hat{y}(t+1)$, thus approximating $J(t+1)$.

Action neural network training is based on model and critic network outputs, by minimizing criterion:

$$\|E_A\| = \sum_t E_A^T(t) \cdot E_A(t) \tag{18}$$

where:

$$E_A(t) = \frac{\partial y(t)}{\partial u(t)} \cdot \frac{\partial J(t)}{\partial y(t)} \tag{19}$$

Partial derivatives $\frac{\partial y(t)}{\partial u(t)}$ and $\frac{\partial J(t)}{\partial y(t)}$ are determined from model network using backpropagation in the manner described in section 2.5.

Minimization of scalar product $\| E_A \|$ in (18) by training of action network results in minimization of Bellman sum in $J(t)$ in (13). Minimum of $J(t)$ is achieved when $\frac{\partial J(t)}{\partial y(t)} \approx 0$, as action network training criterion approaches zero, thus ensuring end of training process.

Action neural network is trained from the output network (unsupervised algorithms). Training of neural networks is carried out using (19) with the training concept shown in Figure 7c.

### 4.2. Dual heuristic programming

Dual Heuristic Programming optimal control algorithm is developed with a goal of increasing procedure efficiency by increasing training speed.

Model network training is the same as in HDP procedure. The improvement is in the direct calculation of the Bellman sum partial derivative over state space variables $\frac{\partial J(t+1)}{\partial y(t)}$ as output of critic network.

Structure of DHP control system is shown in Figure 8.

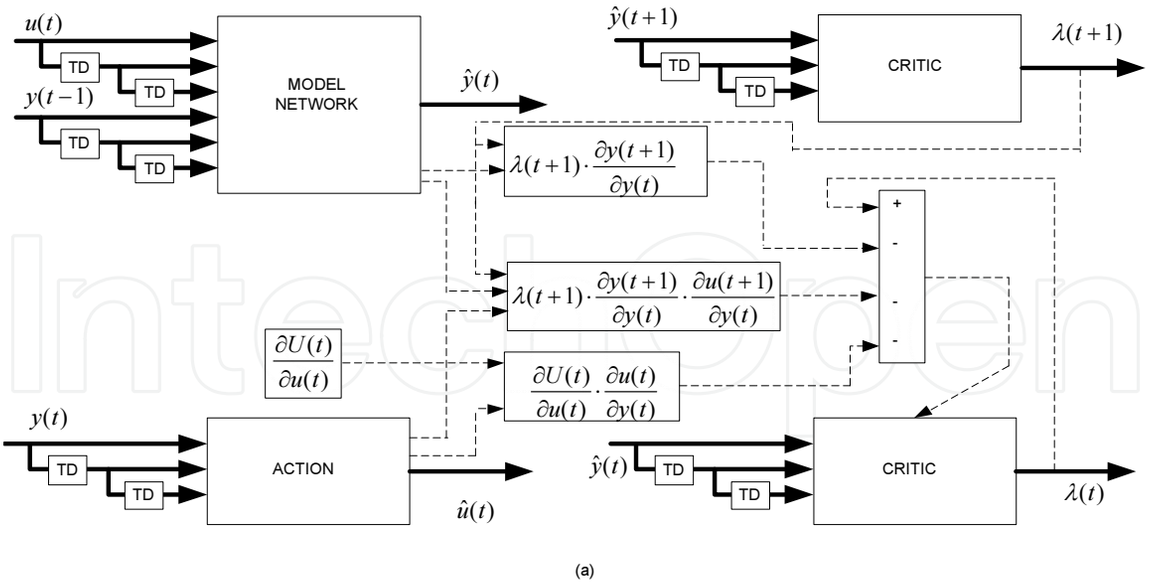
Training of critic neural network minimizes scalar product:

$$\|E_C\| = \sum_t E_C^T(t) \cdot E_C(t) \tag{20}$$

with

$$E_C(t) = \frac{\partial J[\hat{y}(t)]}{\partial \hat{y}(t)} - \gamma \frac{\partial J[\hat{y}(t+1)]}{\partial y(t)} - \frac{\partial U[y(t)]}{\partial y(t)} \tag{21}$$

where $\frac{\partial J[\hat{y}(t)]}{\partial \hat{y}(t)}$ represents partial derivative of Bellman sum over state space variables, and is estimated from critic network. The second part of (21) is determined by deriving



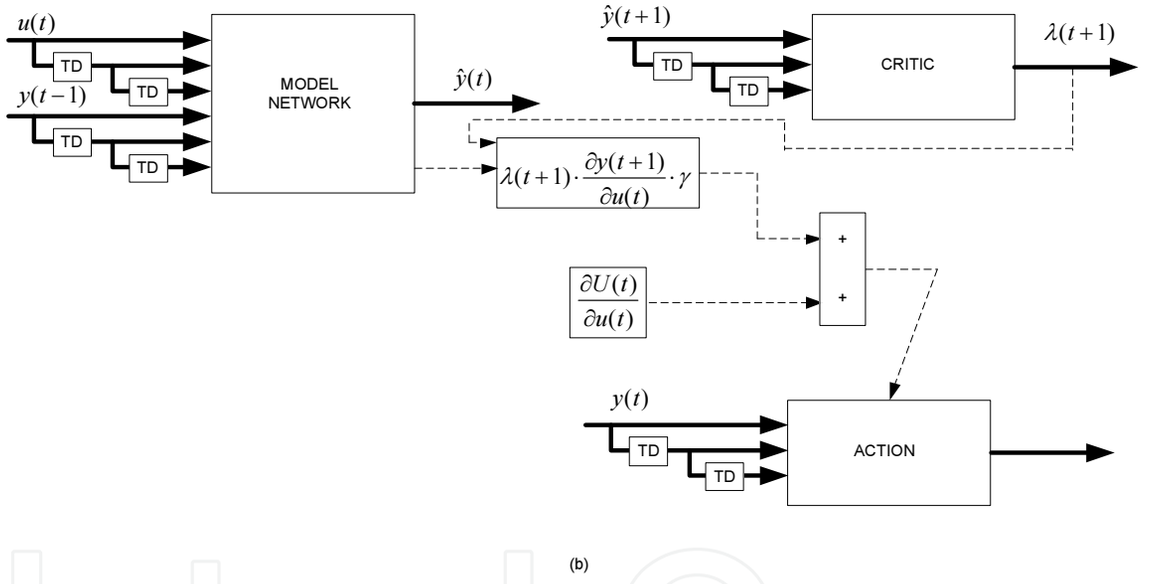




**Figure 8.** DHP structure, (a) Critic NN training, (b) action NN training

$$\frac{\partial J(t+1)}{\partial \hat{y}_j(t)} = \sum_{i=1}^{n} \lambda_i(t+1) \frac{\partial \hat{y}_i(t+1)}{\partial \hat{y}_j(t)} + \sum_{k=1}^{m} \sum_{i=1}^{n} \lambda_i(t+1) \frac{\partial \hat{y}_i(t+1)}{\partial \hat{u}_k(t)} \frac{\partial \hat{u}_k(t+1)}{\partial \hat{y}_j(t)} \tag{22}$$

where

$\hat{\lambda}(t+1) = \frac{\partial J[\hat{y}(t+1)]}{\partial \hat{y}(t+1)}$,

$n$ is size of model network output vector $\hat{y}(t)$,

$m$ is size of control vector $\hat{u}(t)$.

Third part of equation (22) is determined from deriving

$$\frac{\partial U(t)}{\partial Y(t)} = \sum_{k=1}^{m} \frac{\partial U(t)}{\partial u_k(t)} \frac{\partial u_k(t+1)}{\partial y_j(t)} \tag{23}$$

Somewhat simplified structure of DHP is shown in Figure 8a. Partial derivatives $\frac{\partial \hat{y}_i(t+1)}{\partial \hat{y}_j(t)}$, $\frac{\partial \hat{y}_i(t+1)}{\partial \hat{u}_k(t)}$, $\frac{\partial U(t)}{\partial u_k(t)}$ are determined from model network, and partial derivatives $\frac{\partial \hat{u}_k(t+1)}{\partial \hat{y}_j(t)}$ from action network. Prediction of states $\hat{y}(t+1)$ and $\lambda(t+1)$ is determined from model network.

Action neural network is trained to minimize Bellman sum, $\min[J(t)]$, from the criterion $\frac{\partial J[\hat{y}(t)]}{\partial \hat{y}(t)} \approx 0$.

Thus, equation (22) can be transformed to

$$\gamma \frac{\partial J\left[\hat{y}(t+1)\right]}{\partial y(t)} + \frac{\partial U\left[y(t)\right]}{\partial y(t)} = 0 \tag{24}$$

Training of Action Network minimizes scalar product:

$$\left\| E_A \right\| = \sum_t E_A^T(t) \cdot E_A(t) \tag{25}$$

with

$$E_A(t) = \gamma \frac{\partial J\left[\hat{y}(t+1)\right]}{\partial y(t)} + \frac{\partial U\left[y(t)\right]}{\partial y(t)} \tag{26}$$

Simplified training procedure of action NN is shown in Figure 8a.

Cross training of critic and action networks results in slower approximation. In order to reduce training time, it is recommended to begin the procedure with action network pre-trained as linear controller.

### 4.3. Global dual heuristic programming

Globalized Dual heuristic optimal control algorithm uses both algorithms described.

Critic neural network provides approximation of Bellman sum $J(t)$ and derivative $\frac{\partial J(t+1)}{\partial y(t)}$

Action network is trained using the same procedure as described for HDP and DHP.

## 5. Neural networks based control of excitation voltage

Main elements of electric power system are generators, transmission and distribution network and consumers. Transmission and distribution network form a power grid, which is usually very large in geographic terms. Grid control is implemented on both local and global level, mostly on the power generation side of the grid. Specific to power grid control is large number of closed control loops governing the power and voltage of synchronous generators. Transmission network is governed by setting voltage on transformers in order to minimize transmission losses. Governing of generator power and voltage achieves balance of power as well as reactive power flow on the transmission system. Voltage control also has a large impact on the dynamic system behavior.

Generator power output is governed on the generator unit. For water and steam turbine generators, it defines local power transmission to the grid, while also influencing system frequency. Considering the inertness of the generator systems, power output is not difficult to govern. Voltage control manages the balance between generator and grid voltage, and determines reactive power flow. A change in generator voltage setpoint has impact on system state variables and parameters. Disregarding system transients, a change of generator voltage causes a change on generator busbars, generator current and reactive power.

If we also consider system dynamics, generator voltage change also changes system parameters influencing system damping. Synchronous generator connected to the power grid is dominantly nonlinear in characteristics. Generator output power is proportional to generator and grid voltage, while the relation to angle between generator and grid voltage is a sinus function. Maximum theoretical output power is realized with a load angle being $\delta = \frac{\pi}{2}$. Exceeding of maximum value of load angle causes large disturbances and disconnection from power grid.

Operating point of generator connected to the grid is determined by steady state excitation current. Excitation voltage and current values dictate generator voltage, reactive power and load angle.

Frequent small disturbances expected and part of normal operating conditions, resulting in oscillations of generator electromechanical variables. Especially pronounced is the swinging of generator power, reaching several percents of nominal value, with a frequency around 1Hz. It is caused by low system damping and outer disturbances on the grid. Also present on the grid are oscillations caused by the system characteristic frequencies, usually around 0.1Hz.

Power grid can be represented as a single machine connected to an infinite bus system. For each operating point a characteristic frequency and damping can be determined.

In case of small disturbances with a synchronous generator working around predefined operating point, system can be considered linear, but only if it returns to the same operating

http://dx.doi.org/10.5772/58377

point after each disturbance. For those conditions, power system stabilizer (*PSS*) is added to AVR. Based on change of output power and load angle, it modifies voltage regulator setpoint and increases system damping. PSS is configured using linear control theory, and is presently a standard part of automatic voltage regulator systems [10].

Large disturbances occur after short circuits on the power grid, transmission and generator outages, and connection and disconnection of large loads.

Electromechanical oscillations occurring as a consequence of disturbances cause damped oscillations of all electrical and mechanical values (generator voltages and currents, active and reactive power, speed and load angle). Oscillations are also propagated by AVR to excitation system values.

Large system disturbances can cause significant changes to local parameters relevant for generator stability. Due to the linear nature of the voltage regulator system, a change in grid voltage or configuration leads to generator working in non optimal state. Even if the system returns to the same operating conditions, passing through nonlinear states during disturbance causes non optimal control, diminishing the stabilizing effects of the voltage regulator on the system. This can in turn cause another disturbance or, in case of larger units, disruption of the whole power grid. Limitations of the use of linear control systems call for an improved system which can provide optimal control of generator excitation system [13]- [15].

There is a lot of research effort directed toward design of excitation control system based on neural networks. However, very few of the demonstrated solutions have been implemented on real plants.

The goal of the chapter is to develop a practical-to-use system. This is mainly achieved by separating the design on preparation and implementation phase. Almost all algorithm steps are executed in off-line mode, up to the final step of digital voltage regulator implementation.

To implement the voltage regulator described in the chapter following equipment is needed:

- data acquisition hardware and software
- personal computer with Matlab and Simulink environment
- automatic voltage regulator with neural network support (no training support is needed)

Described controller is not adaptive. Adaptive critic design (ACD) algorithm is used for optimal control.

Classic voltage regulators (AVR) equipped with PSS can be very effective in voltage control and power system stabilization if it is used in well configured transmission grid. In such conditions the generator stays in the linear operating range and the controller maintains optimal configuration on change of active and reactive power. Voltage regulator accuracy, balance of reactive power, stabilizing effect during large disturbances and active stabilization in normal operating conditions are achieved.

If the generator is connected to relatively weak transmission grid, with possible configuration changes during operation, use of neural network based control of excitation voltage leads to better results.

### 5.1. HDP algorithm implementation

In order to achieve optimal control of synchronous generator connected to the power grid, it is necessary to select a minimum of two system variables – generator voltage and either output power or angular velocity. Selected variables must reflect system oscillations during disturbances. It is also necessary to choose cost function that is minimized during training. System optimization is achieved by minimization of squared generator voltage governing error $\Delta U(t)=U_{G\,REF}-U_G$ and minimization of output power change $\Delta P$ obtained from:

$$U(t)=\left(K_{u1}\cdot\Delta U_G(t)+K_{u2}\cdot\Delta U_G(t-1)+K_{u3}\cdot\Delta U_G(t-2)\right)^2+\left(K_{P1}\cdot\Delta P(t)+K_{P1}\cdot\Delta P(t-1)+K_{P1}\cdot\Delta P(t-2)\right)^2 \tag{27}$$

It is important to notice that the two demands are in conflict. Increase in gain in main control loop reduces static control error, but also reduces system damping and increases power output deviation $\Delta P$. Use of cost function (27) ensures training process that increases system damping without sacrificing controller responsiveness.

System identification is carried by training model neural network. Figure 8 shows training process of model neural network. Identification of nonlinear system is made by series-parallel structure.

For ACD algorithm optimal control two output state space variables are chosen – generator voltage and output power, $y=[\Delta U_G;\Delta P]$. Vector $y$ is used to form the cost function. Input vector is

$$u(t)=\left[\,y_r(t)\;y_r(t-1)\;y_r(t-2)\;y(t-1)\;y(t-2)\;y(t-3)\right]^T \tag{28}$$

where

$y_r=U_{GREF}$ represents generator voltage setpoint,

$u_f$ – exciter control signal,

$y=[\Delta U_G\;\Delta P\,]^T$ is system output.

Model NN has 9 inputs and 2 outputs.

Figure 9 shows simulation block diagram of identification procedure, implemented in Matlab and Simulink environment.

http://dx.doi.org/10.5772/58377

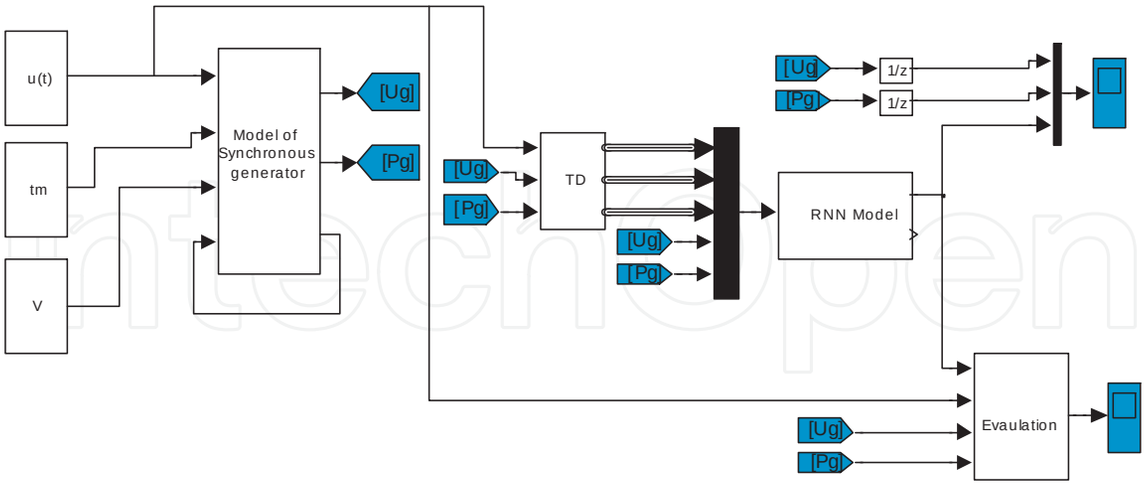



**Figure 9.** Model identification block diagram

A third order nonlinear mathematical model was used as a synchronous generator [20]. Synchronous generator is modeled as a single machine connected to an infinite bus. Although the model ignores initial magnetic flows, it proves to be a good approximation of an actual generator. A predictor-corrector method is used to solve differential equations.

Selected mathematical model, used with well conditioned state space variables and described methods, achieves good numerical stability during simulation, allowing for relatively large sample time ($T_D$=0.02s). Described model of synchronous generator is implemented in C-language S-function and used for further simulations.

For model identification, recurrent neural network is used, with five neurons in hidden layer and two neurons in output layer. Hidden layer is implemented with full recursion and *tansig* activation function. Output layer uses linear activation function.

Initial learning rate value $\eta$ (21) is set to $\eta$=$10^5$. It was reduced during training to $\eta$=$10^2$. Training was completed in ~10000 steps. Results are shown in Figure 10. Signals $U_G$ NN(0) and $U_G$ NN(1) represent generator voltage at the start and at the end of RNN training. Figure 10a compares responses of control error of generator model and neural network. Voltage setpoint is generated as a pulse signal with a magnitude of ± 10%. In figure 10b, response to the change of generator power is shown. Figure 10c shows squared sum of RNN error during training.

In the same experiment efficiency of the algorithm was verified. Training was stopped after 100000 steps. For the rest of the simulation, network output is calculated without the change of weight coefficient, thus eliminating the recency effect.

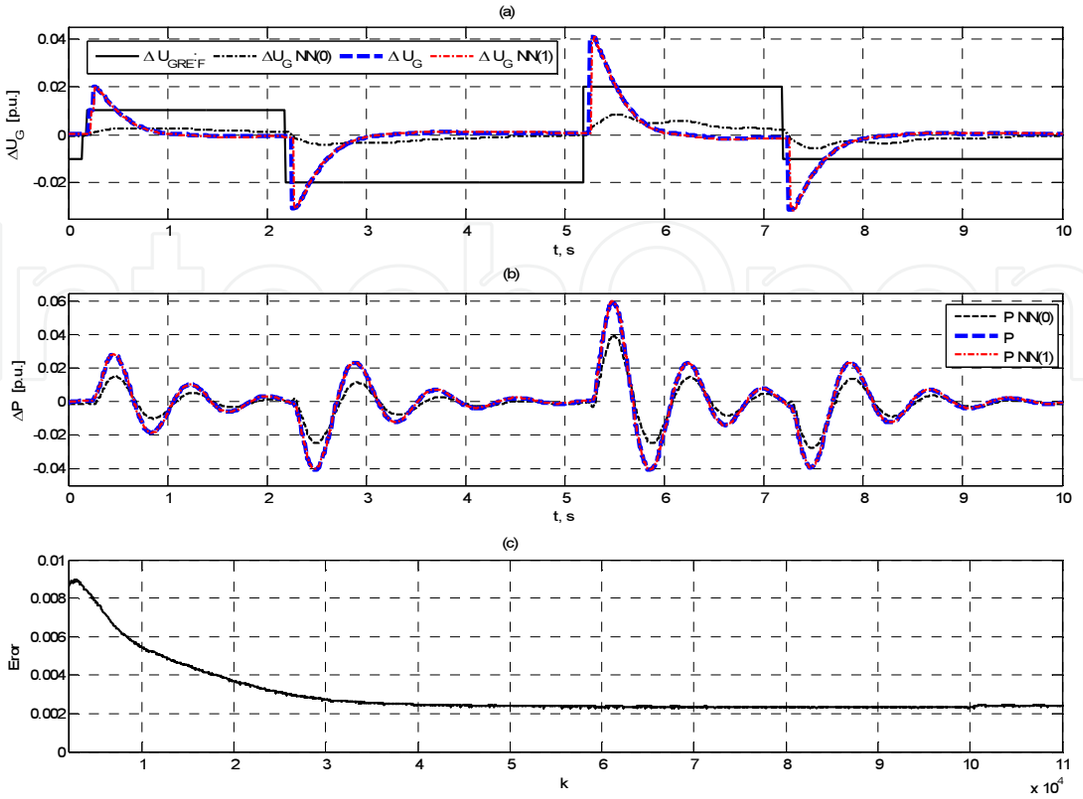



**Figure 10.** Training results of model neural network

Once the model neural network is trained, it is possible to create simulation for training critic and action neural networks. Simulink model of the simulation is shown in Figure 11.

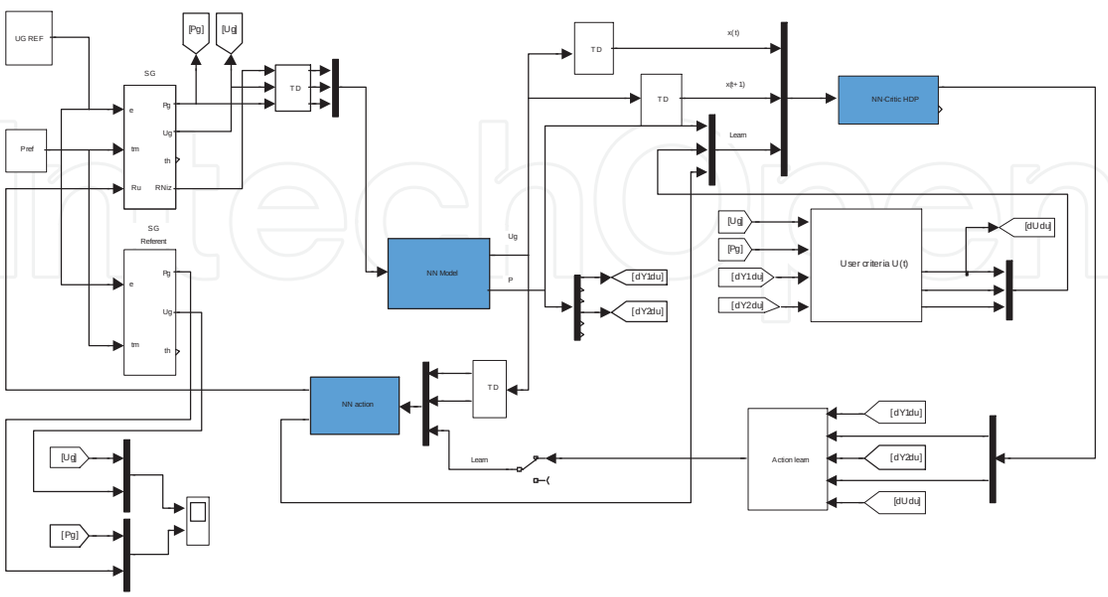



**Figure 11.** Simulink model for training action and critic networks

Simulation uses expressions (21), (22) and (23).

Neural network training is implemented in Matlab S-functions, written in C programming language.

Input of the S function is formed of system signals and desired value. S-function returns RNN output signals and partial derivatives of outputs $\left(\frac{\partial y_i(t)}{\partial u_j(t)}\right)$. Critic and action networks are trained in parallel.

Training results are shown in Figure 12. Figure 12a compares generator voltage response using conventional automatic voltage regulator (AVR) to neural network based controller trained using HDP algorithm. It can be seen that voltage rate of change was not degraded as a result of system damping. It is important to emphasize that generator voltage rate of change is key to system stability during large disturbances.

Figure 12b compares active power output oscillation of conventional and HDP controller. Figure 12c shows load angle, which is indicative of system stability $\left(\delta < \frac{\pi}{2}\right)$. System damping is significantly increased by use of neural network controller.

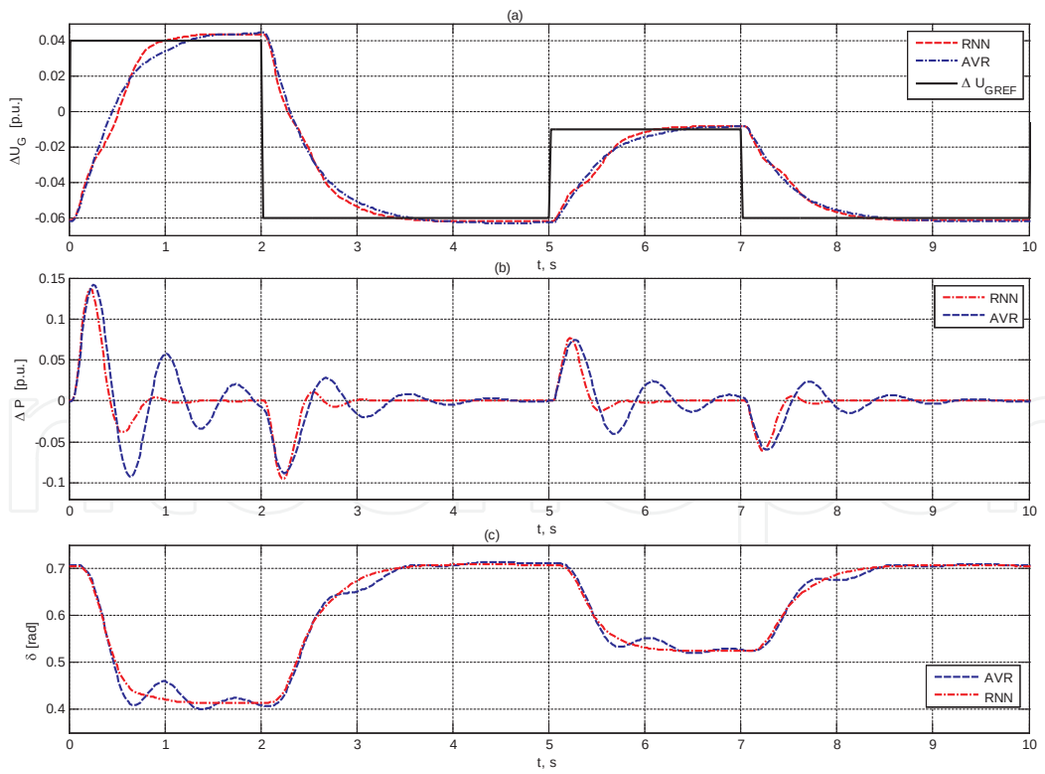



**Figure 12.** Response comparison of conventional AVR and HDP controller. a) generator voltage; b) generator active power; c) load angle

HDP algorithm is also verified on voltage control of synchronous generator from Simulink SimPower System block library (Synchronous Machine).

Model network is designed with one hidden recursion layer with 5 neurons. Critic and action networks use the same structure, but with four neurons in hidden layer.

HDP algorithm shown in Figure 11 is applied. Training of neural networks is performed in two steps– in first step action NN is trained to mimic classic voltage regulator, and in second step DHP algorithm is used to train both action and critic NN. It is important to get initial values of action NN in first step, as DHP needs network partial derivatives in training critic NN. Figure 13 shows training results.

Figure 13a shows comparison of classic and NN controller responses. Generator voltage rate of change is almost the same for both controllers.

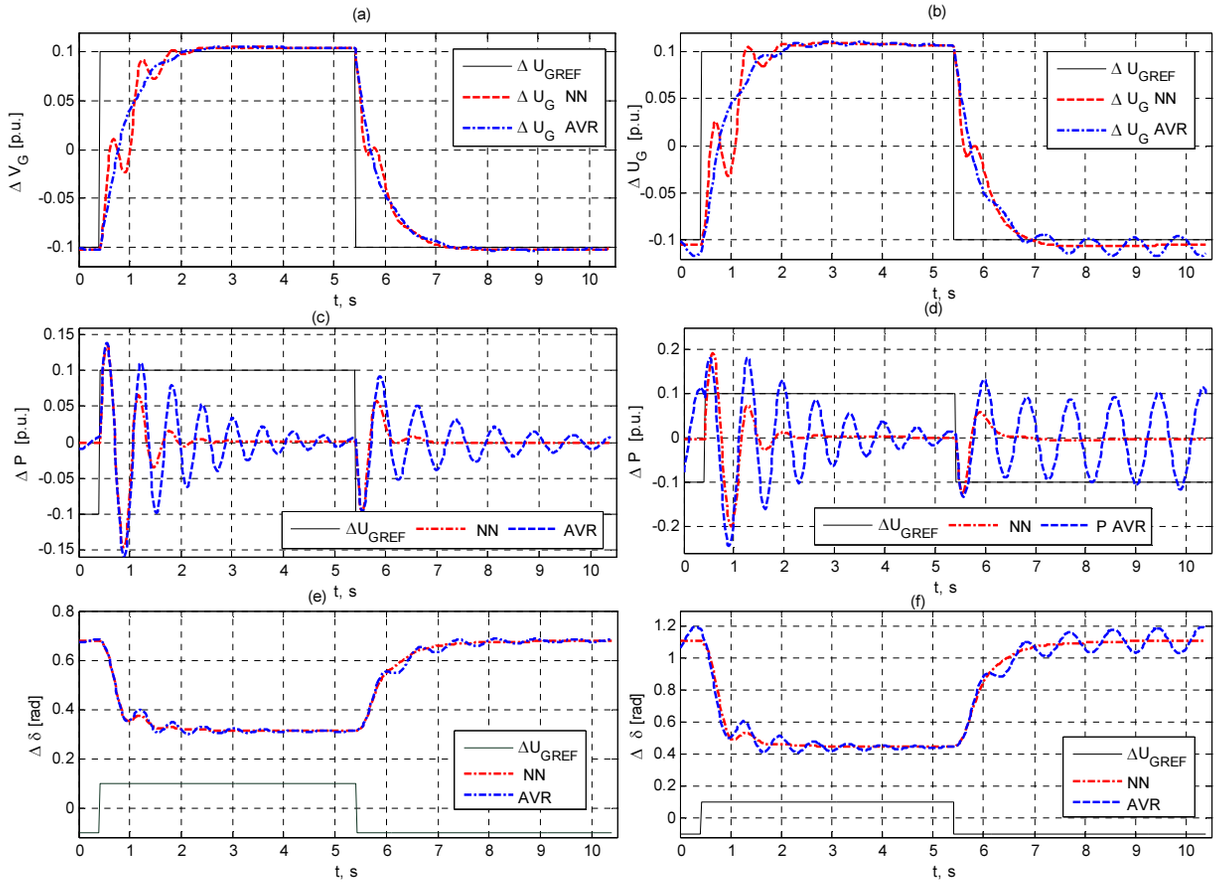



**Figure 13.** Response comparison of conventional AVR and HDP controller a), b) generator voltage; c), d) generator active power; e), f) load angle

Figure 13c and 13e show comparison of output power and load angle change. It is obvious that the HDP neural network controller shows significant improvement over classic regulator.

Figure 13b shows comparison of voltage transient in the condition generator voltage reduced to 90% of nominal value. Rate of change of generator voltage is maintained even during such extreme operating conditions.

Figure 13d shows conventional automatic voltage regulator (AVR) operating on the edge of stability, while the DHP controller is completely stable. Comparison of load angle values demonstrates significantly better behavior of DHP controller. On large load angles DHP controller maintains stability while conventional AVR provides low system damping, leading to instability and possible outages.

Neural network HDP voltage controllers were developed using DHP algorithm. Experimental results showed that DHP algorithm is more efficient than ADC algorithm. Implementation of GDHP does not bring significant improvements over DHP.

Application of developed S-functions enables simple use of described algorithms, as well as experimenting with different function parameters.

## 6. Real time windows target implementation

DHP optimal control algorithm has also been implemented using Matlab Real Time Windows Target Toolbox (RTWT), based on [18] and [19]. Simulation hardware consists of desktop PC equipped with National Instruments data acquisition card NI DAQ 6024.

Use of RTWT platform enables direct implementation of developed controllers on the real system.

In Figure 14, a system consisting of synchronous generator and RTWT controller is shown. Signal acquisition is achieved using DAQ 6062E Input, with AD conversion of the input values. Line voltage and phase current signals are processed in "Clark" S-function, using Clarke transformation. Transformed values of generator voltage $U_G$ and generator power $P$ are used as control feedbacks. Digital to analog conversion is performed by output module DAQ 6062 Output. All simulation blocks are part of a standard Simulink library except Clark transformation and RNN S-functions.

ACD algorithm is implemented in two steps. Classic voltage controller is used in the first step (Figure 14). In the second step it is replaced with NN controller.

During testing, generator active power and voltage were changed for a wide range of values. A pulse signal was superimposed to the generator voltage set-point value, resulting in large disturbances on the generator. To obtain measurement records RTWT Signal Visualization was used, with measured data saved in Workspace Simulink block *yP*. Recorded data was used to train neural network in offline mode.

A RNN was used with one hidden layer containing 5 tansig neurons. Model NN had 9 inputs and 2 outputs. Results of NN training are displayed in Figure 15.

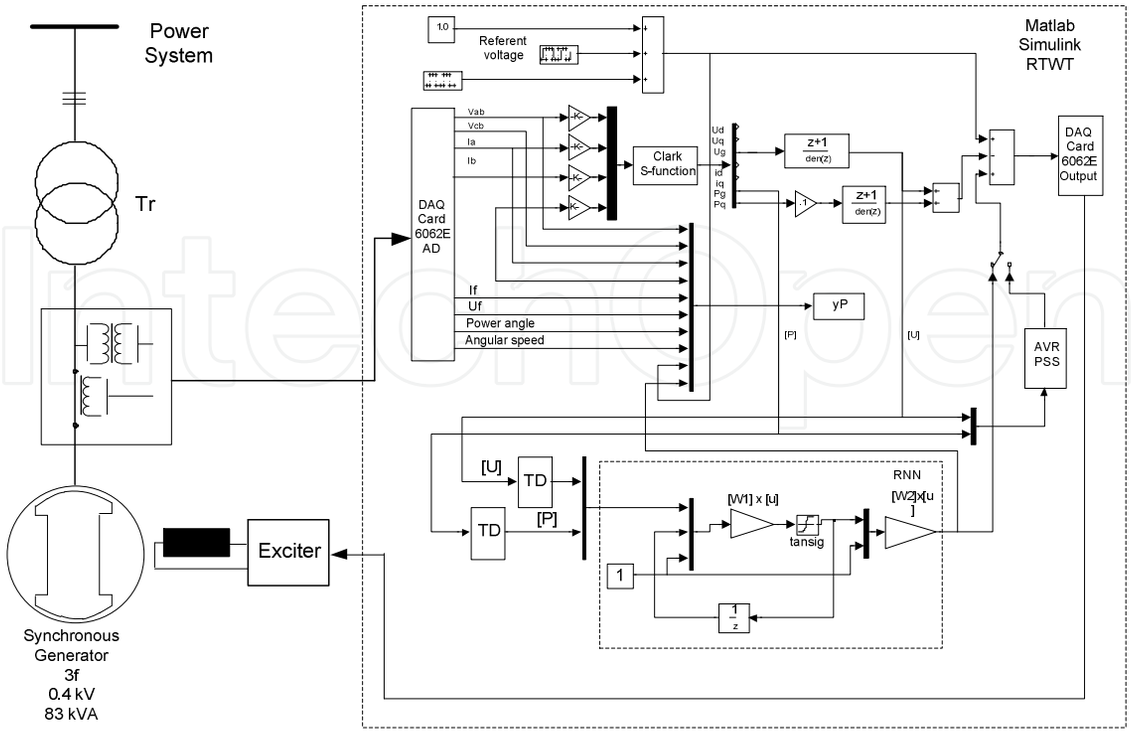



**Figure 14.** Automatic voltage control of synchronous generator using RTWT platform

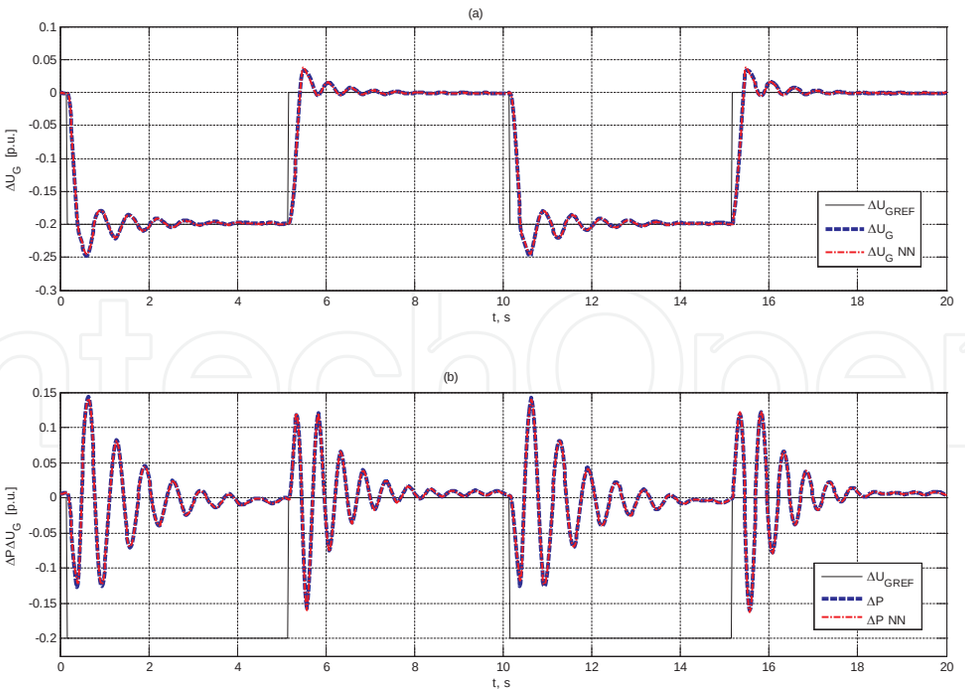



**Figure 15.** Results of Model NN training a) generator voltage; b) generator active power

http://dx.doi.org/10.5772/58377

Obtained model NN was used to train critic and action RNN using procedure described in 3. The trained action NN was transferred to Simulink block (Figure 14) and prepared for use in real time. Using RTWT module system was run in real-time mode, with trained action RNN as a voltage controller.

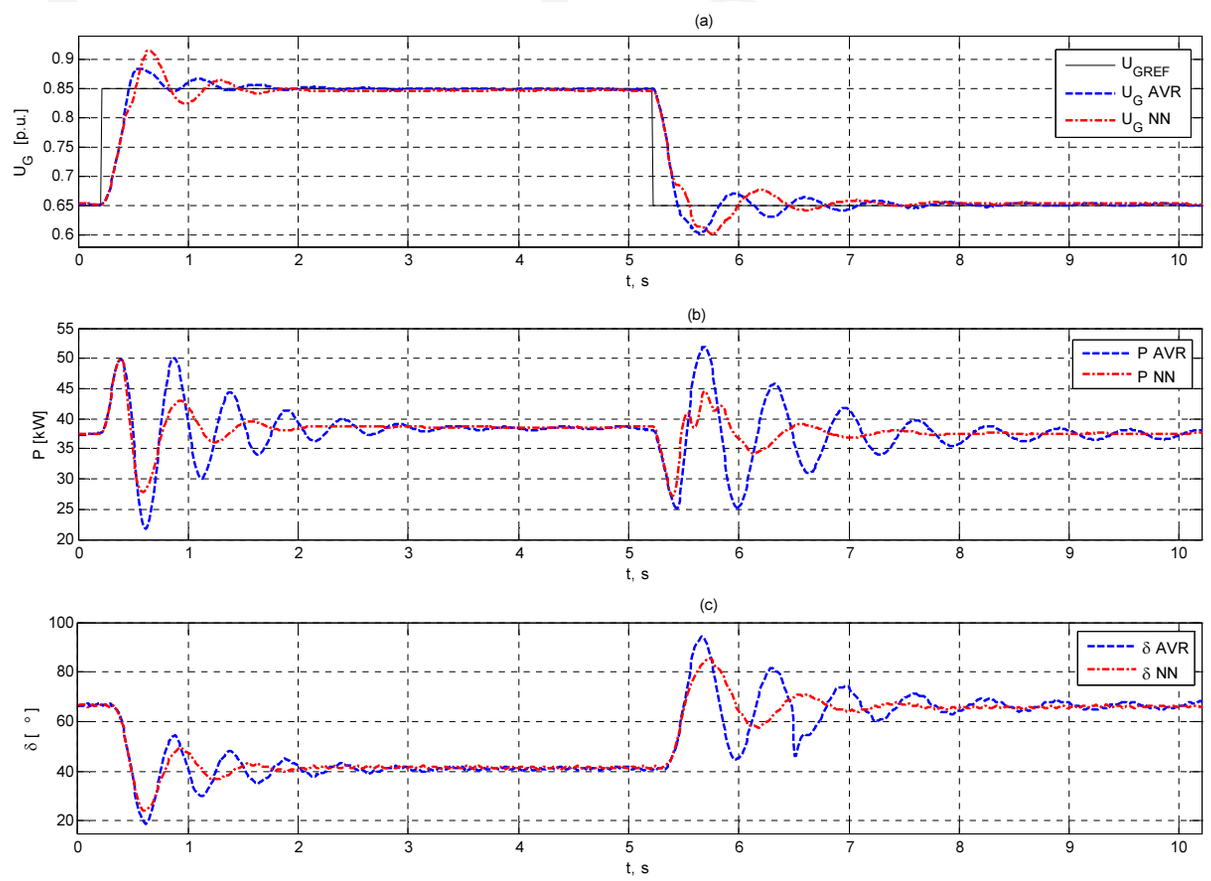



**Figure 16.** Classic controller and HDP NN controller – system responses: a) generator voltage; b) generator active power; c) load angle

Simulation results are shown in Figure 16, comparing classic and DHP voltage controllers. Figure 16a represents comparison between generator voltage responses and shows satisfactory rate of generator voltage change and high control accuracy. In Figure 16b, generator active power outputs are compared. Considerable damping was achieved using DHP controller. Improved system stability during large disturbances is visible in Figure 16c, where the maximum load angle deviation is significantly lower for DHP controller.

RTWT controller was implemented in sample-time mode with sample time set to $T_s = 2.5 \cdot 10^{-4} s$. Control loop sample time is $T_D = 0.02 s$.

Matlab Simulink RTWT Toolbox simplifies the transfer of developed procedures to the hardware platform.

## 7. Conclusion

In the chapter, a complete process of development and implementation of neural network based controller is described. Optimal control is achieved by training neural network using data obtained from real system. Developed algorithms were tested on voltage control of synchronous generator connected to the power grid.

Optimal control algorithm is implemented in Matlab and Simulink environment. Neural network training is implemented in C programming language Matlab S-function. S-function execution time was sufficiently low to achieve real time performance.

Developed algorithm is tested using Matlab Real Time Windows Target Toolbox, using the same models with minimal modifications.

Experiments performed on laboratory system show possibility of building optimal controller based on special network training procedure with no need for adaptive features.

Obtained simulation results show advantages of neural networks based controller over classic controllers. Developed procedure can relatively easily be adapted for use on real systems.

## Appendices

### Appendix A

Matlab function for calculating hidden layer derivatives of outputs over weight coefficients of hidden layer.

```
function[dadWR,a]=dadWR(WW1,WW2,p,a_1,dadWR_1,NW1,NW2);
%function determines the partial derivative  output recursive hidden%layers to
weight coefficients matrix WW1% WW1 matrix of hidden layer% WW2 matrix of out-
put layer% NW1 number of neurons in the hidden layer% NW2 length of the output
vector neural networks% Ni  length of the input vector neural networks% J  tar-
get output value of the neural network% a output value of hidden layer neurons
% p  input vector of the hidden layer of recurrent neural network

Ni=length(p)-NW1;
a_temp=WW1*[p; 1];
a=tansig(a_temp);
Ia=ones(NW1,1)-(a.*a);  %value of derivative tansig activation function of neu-
rons -
%y=WW2*a;
%J=J-y;
for l=1:NW1
    for i=1:NW1
        for j=1:Ni+NW1
            dadar_temp=0;
            for k=1:NW1;
                dadar_temp=dadar_temp+WW1(l,Ni+k)*dadWR_1(k,(i-1)*Ni+j);
```

```
            end
            dadWR(l,(i-1)*(Ni+NW1)+j)=Ia(i)*((l==i)*p(j)+dadar_temp);

        end,
    end,
end,

%end  function dadWR
```

## Appendix B

S-function for RNN training with one hidden layer.

The first part defines sample time, number of inputs, outputs and neurons. Input is formed of values used for training. Output 1 is RNN output. Output 2 is partial derivative of output over input. Training is performed using Kalman filter.

```
/*  File    : model.c *  Abstract: *  For more details about S-functions, see
simulink/src/sfuntmpl_doc.c. *  Copyright 1990-2002 The MathWorks, Inc. *  $Re-
vision: 1.12 $ */
                        /* ================model==================== * This s-
function is used for learning recurrent neural network (RNN) in programming en-
vironment Matlab Simulink * Learning RNN is implemented using the Kalman
filter * This program supports RNN with one hidden layer * In the first part
defines the time discretization, number of inputs, number of outputs and * the
number of neurons in a single hidden layer * Simultaneously with learning RNN
this function account the partial derivative of output per inputs * Port num-
ber of the s-function is equal to the number of inputs in RNN + number of out-
puts * Mato Miskovic Imotica */#define S_FUNCTION_NAME model
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"
#include <math.h>
#include <string.h>

#define TS 1    //sample time
#define N_ULD 3 //number of inputs
#define N_TD 1  //number of steps backwards (t-1), (t-3),(t-3), (t-N_TD)
#define NUL    (N_ULD*N_TD)
#define NIZ 1   //number outputs
#define W1 5    // number of neurons in the hidden layer
#define W1_j   (NUL+W1+1)   //
#define W2_j   (W1+1)
#define W1_ij  (NUL+W1+1)*W1
#define W2_ij  NIZ*W2_j
#define N_W1   0
#define N_W2   N_W1+W1_ij
#define SIZE_P  (W1_ij+W2_ij)
#define N_P    N_W2+W2_ij
#define NH_1   N_P+ SIZE_P*SIZE_P
#define N_dadWR NH_1+W1
#define N_OUT   N_dadWR+W1*W1_ij
#define N_dnet  N_OUT+NIZ
```

```
#define N_ON    N_dnet+NIZ*N_ULD
#define NX      N_ON+10 //
#define ETTAMI 10.0   //final value of the   learning rate
#define ETTAQ .051    // Q
#define ITER_UCI 20000  //number of steps to stop learning RNN
#define NDISCSTATES NX

static
                    void ttsig(real_T x, real_T *y)  //tansig
{
    *y=2.0/(1.0+exp(-2.0*x))-1.0;
}
//AxB=C multiplying matrices in line form
static
                      void aputab_c(real_T *aa, real_T *bb, real_T *cc, int_T
a_r, int_T a_c, int_T b_c){
    int_T i, j, k;
    real_T priv;
    for(i=0; i<a_r  ;i++){
        for(j=0;j<b_c;j++){
            priv=0;
            for(k=0;k<a_c;k++)  {
                priv=priv+aa[k+i*a_c]*bb[j+b_c*k];
            }
            cc[i*b_c+j]=priv;
        }
    }
}
//XxB=C multiplying matrices in line form,  x from SimStruc
static
                      void xputab_c_pok(SimStruct *S, real_T *bb, real_T *cc,
int_T a_r, int_T a_c, int_T b_c, int_T x_p){
    int_T i, j, k;
    real_T priv;
    real_T *x       = ssGetRealDiscStates(S);
    for(i=0; i<a_r  ;i++){
        for(j=0;j<b_c;j++){
            priv=0;
            for(k=0;k<a_c;k++)  {
                priv=priv+x[x_p+k+i*a_c]*bb[j+b_c*k];
            }
            cc[i*b_c+j]=priv;
        }
    }
}
//AxX=C multiplying matrices in line form,  x from SimStruc
static
                      void aputax_c_pok(SimStruct *S, real_T *aa, real_T *cc,
int_T a_r, int_T a_c, int_T b_c, int_T x_p){
    int_T i, j, k;
    real_T priv;
    real_T *x       = ssGetRealDiscStates(S);
```

```
    for(i=0; i<a_r  ;i++){
        for(j=0;j<b_c;j++){
            priv=0;
            for(k=0;k<a_c;k++)  {
                priv=priv+aa[k+i*a_c]*x[x_p+j+b_c*k];  }
            cc[i*b_c+j]=priv;
        }
    }
}
//AxB=X multiplying matrices in line form,  x to SimStruc
static
                    void aputab_x_pok(SimStruct *S, real_T *aa, real_T *bb,
int_T a_r, int_T a_c, int_T b_c, int_T x_p){
    int_T i, j, k;
    real_T priv;
    real_T *x       = ssGetRealDiscStates(S);
    for(i=0; i<a_r  ;i++){
        for(j=0;j<b_c;j++){
            priv=0;
            for(k=0;k<a_c;k++)  {
                priv=priv+aa[k+i*a_c]*bb[j+b_c*k];
            }
            x[x_p+i*b_c+j]=priv;
        }
    }
}
static
                    void transp(real_T *aa, real_T *bb, int_T a_r, int_T a_c)
{ //transposed matrix
    int_T i, j;
    for(i=0; i<a_c  ;i++){
        for(j=0; j<a_r  ;j++){
            bb[i*a_r+j]=aa[i+j*(a_c)];  }}
}
static
                    void invmatrix(real_T *aa, int_T n){ // inverse matrixin
inline form
    int_T l, m, p;
    int_T pos, p_old;
    int_T p_old2;
    for(l=0;l<n;l++){
        p_old2=l*n+l;
        aa[p_old2]=1/(aa[p_old2]);
        for(m=0;m<n;m++)
            if(m!=l){
            p_old=m*n+l;
            aa[p_old]=aa[p_old]*aa[p_old2];
            }
        for(m=0;m<n;m++){
            for(p=0;p<n;p++){
                if(m!=l){
                    if(p!=l){
```

```
                        pos=m*n+p;
                        p_old=m*n+l;
                        p_old2=l*n+p;
                        aa[pos]=aa[pos]-(aa[p_old]*aa[p_old2]);
                    }
                }
            }
        }

        p_old=l*n+l;
        for(p=0;p<n;p++)
            if(p!=l){
            pos=l*n+p;
            aa[pos]=-aa[p_old]*aa[pos];
            }
    }
    pos=n*n;
}
/*====================*
 * S-function methods *
 *====================*/
static
                    void mdlInitializeSizes(SimStruct *S) {
    ssSetNumSFcnParams(S, 0);  /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter mismatch will be reported by Simulink */
    }
    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, NDISCSTATES);
    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, NUL+NIZ);
    if (!ssSetNumOutputPorts(S, 2)) return;
    ssSetOutputPortWidth(S, 0, NIZ);
    ssSetOutputPortWidth(S, 1, NIZ*N_ULD);
    ssSetNumSampleTimes(S, 1);
    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
    ssSetNumPWork(S, 0);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
}
static
                    void mdlInitializeSampleTimes(SimStruct *S) {
    ssSetSampleTime(S, 0, TS);
    ssSetOffsetTime(S, 0, 0.0);
}
#define MDL_INITIALIZE_CONDITIONS
// Function: mdlInitializeConditions ========================================
static
                    void mdlInitializeConditions(SimStruct *S) {
    real_T *x0 = ssGetRealDiscStates(S);
```

http://dx.doi.org/10.5772/58377

```
    int_T  i, j;
    real_T  tempRAND[22]={0.66, 0.01, 0.42, -0.14, -0.39, -0.62, -0.61, 0.36,
-0.39, 0.08, -0.69, 0.39, -0.24, 0.72, 0.7, 0.18, -0.01, 0.79, 0.64, 0.28,
0.6359, 0.32};
    for (i=0; i<W1_ij+W2_ij; i++){
        x0[i]=.1*tempRAND[i-(i/20)*20];}
    for (i=0; i<SIZE_P; i++){
        for (j=0; j<SIZE_P; j++){
            if (i==j){
                x0[N_P + i*SIZE_P+j]=1000;
            }
        }
    }
}
// Function: mdlOutputs =======================================================


static
                    void mdlOutputs(SimStruct *S, int_T tid) {
    real_T            *Y    = ssGetOutputPortRealSignal(S, 0);
    real_T            *DY   = ssGetOutputPortRealSignal(S, 1);
    real_T            *x    = ssGetRealDiscStates(S);
    int_T i;

    //InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
    UNUSED_ARG(tid); /* not used in single tasking mode */

    for (i=0;i<NIZ;i++)
        Y[i]=x[N_OUT+i];
    for (i=0; i<NIZ*N_ULD; i++)
        DY[i]= x[N_dnet+i];
}
#define MDL_UPDATE
//Function: mdlUpdate ========================================================
static
                    void mdlUpdate(SimStruct *S, int_T tid) {
    int_T  i, j, k, priv_i;
    real_T priv;
    real_T   WW1[W1][W1_j];
    real_T   WW2[NIZ][W2_j];
    real_T   a_pr[W1]={0};
    real_T   a[W1+1]={0};
    real_T   JJ[NIZ]={0};
    real_T   W1R[W1*W1]={0};
    real_T   oout[NIZ];
    real_T   WW2pr[NIZ*W1]={0};
    real_T   dadWR[W1*W1_ij];
    real_T   I_a[W1];
    real_T   PP[W1*W1_ij];
    real_T   dYdW[NIZ*SIZE_P]={0};
    real_T   dYdW_T[NIZ*(SIZE_P)];
    real_T   h_WxP [NIZ*SIZE_P];
```

```
    real_T   K[SIZE_P*NIZ];
    real_T   Ppr[SIZE_P*SIZE_P];
    real_T   Kxh_W[SIZE_P*SIZE_P];
    real_T   pu[W1_j];
    real_T   Apr[NIZ*NIZ]={0};
    real_T   Pxh_W[SIZE_P*NIZ] ;
    real_T   NNx[SIZE_P] ;
    real_T   dnet[NIZ*N_ULD]={0};
    real_T   etttami=0;
    int_T    SIZE_PxSIZE_P=SIZE_P*SIZE_P;
    int_T    W1xW1_ij=W1*W1_ij;
    real_T            *x      = ssGetRealDiscStates(S);
    InputRealPtrsType uPtrs    = ssGetInputPortRealSignalPtrs(S, 0);
    x[N_ON]=x[N_ON]+TS;
    pu[NUL+W1]=1.0;
    for (i=0; i<NUL; i++){
        pu[i]=*uPtrs[i];
    }
    for (i=0; i<W1; i++){
        pu[NUL+i]=x[NH_1+i];
    }
    pu[NUL+W1]=1.0;
    for (i=0; i<NIZ; i++){
        JJ[i]=*uPtrs[NUL+i];
    }
    for (i=0; i<W1; i++) {
        for (j=0; j<W1_j; j++) {
            WW1[i][j]=x[N_W1+i*W1_j+j];
        }
    }
    for (i=0; i<NIZ; i++) {
        for (j=0; j<W2_j; j++) {
            WW2[i][j]=x[N_W2+i*W2_j+j];
        }
    }
    for (i=0; i<W1; i++) {
        for (j=0; j<W1; j++) {
            W1R[i*W1+j]=WW1[i][NUL+j];
        }
    }
     xputab_c_pok(S, pu, a_pr, W1, W1_j, 1, 0);//xputab_c_pok(S,bb, cc, a_r,
a_c,  b_c, x_p)
    for (i=0; i<W1; i++)  {             //a
        ttsig(a_pr[i], &a[i]);      // ttsig(real_T x,real_T *y)
        x[NH_1+i]=a[i];
    }
    a[W1]=1.0;
    for (i=0; i<W1; i++)
        I_a[i]=1.0-a[i]*a[i];
    xputab_c_pok(S, a, oout, NIZ, W2_j, 1, N_W2);          //Y=WW2*[a;1];
    for (i=0; i<NIZ; i++){
        x[N_OUT+i]=oout[i];
```

http://dx.doi.org/10.5772/58377

```
    JJ[i]=JJ[i]-oout[i];
}
for (i=0; i<W1*W1_ij; i++)
    PP[i]=0;
for (i=0; i<W1_j; i++)
    PP[i]=pu[i];
for (i=1; i<W1; i++){
    for (j=0; j<W1_j; j++){
        PP[i*(W1_ij+W1_j)+j]=pu[j];
    }
}
//dnet
for (i=0; i<NIZ; i++){
    for (j=0; j<N_ULD; j++){
        priv=0.0;
        dnet[i*N_ULD+j]=0.0;
        for (k=0; k<W1; k++){
            priv=priv+WW2[i][k]*(1.0-(a[k]*a[k]))*WW1[k][j*N_TD];}
        x[N_dnet+i*N_ULD+j]=priv;
    }
}
aputax_c_pok(S, W1R, dadWR, W1, W1, W1_ij, N_dadWR);
for (i=0; i<W1xW1_ij; i++)
    PP[i]=PP[i]+0.9*dadWR[i];
for(i=0;i<W1;i++){
    for(j=0;j<W1_ij;j++){
        x[N_dadWR+i*W1_ij+j]=I_a[i]*PP[i*W1_ij+j];
    }
}
for(i=0;i<NIZ;i++){
    for(j=0;j<W1_ij;j++){
        priv=0.0;
        for(k=0;k<W1;k++){
            priv=priv+x[N_W2+i*W2_j+k]*x[N_dadWR+k*W1_ij+j];
        }
        dYdW[i*SIZE_P+j]=priv;
    }
}

for(i=0; i<NIZ; i++) {                                  // dydW2
    for(j=0; j<W2_j; j++){
        dYdW[W1_ij+i*(W1_ij+W2_ij+W2_j)+j]=a[j];
    }
}
//Kalman
transp(dYdW, dYdW_T, NIZ, SIZE_P); //dYdW_T=dYdW'
aputax_c_pok(S, dYdW, h_WxP, NIZ, SIZE_P, SIZE_P, N_P);
aputab_c(h_WxP, dYdW_T, Apr, NIZ, SIZE_P, NIZ);
if ( x[N_ON] > ITER_UCI )
    etttami=1e90*ETTAMI;
else etttami=ETTAMI+1000*ETTAMI/(x[N_ON]+1.0);
for (i=0; i<NIZ*NIZ; i=i+NIZ+1){
```

```
            Apr[i]=Apr[i]+etttami;
        }
        invmatrix(Apr, NIZ);
        xputab_c_pok(S, dYdW_T, Pxh_W, SIZE_P, SIZE_P, NIZ, N_P);
        aputab_c(Pxh_W, Apr, K, SIZE_P, NIZ, NIZ);      //K=PM*h_WM*A;
        aputab_c(K, JJ, NNx, SIZE_P, NIZ, 1);
        for(i=0; i<SIZE_P; i++){
            x[i]=x[i] +NNx[i];
        }
        aputab_c(K, dYdW, Kxh_W, SIZE_P, NIZ, SIZE_P); //K*h_WM
        aputax_c_pok(S, Kxh_W, Ppr, SIZE_P, SIZE_P, SIZE_P, N_P);
         for (i=0; i<SIZE_P; i++){     Ppr[i*(SIZE_P+1)]=Ppr[i*(SIZE_P+1)] - ETTAQ/
(x[N_ON]+1.0);
        }
        for (i=0; i<SIZE_PxSIZE_P; i++)
            x[N_P+i]=x[N_P+i]-1*Ppr[i];
        UNUSED_ARG(tid);
    }
    // mdlTerminate
    static
                        void mdlTerminate(SimStruct *S) {
        int_T i, j;
        real_T             *x       = ssGetRealDiscStates(S);
        printf("\n");
        for(i=0; i<W1*W1_j+NIZ*W2_j; i++){
            printf("%f\t", x[i]);
        }
        UNUSED_ARG(S);
    }

    #ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
    #include "simulink.c"      /* MEX-file interface mechanism */
    #else
    #include "cg_sfun.h"       /* Code generation registration function */
    #endif
```

## Author details

Mato Miskovic[1], Ivan Miskovic[2] and Marija Mirosevic[3]

1 HEP Hydro-power Dubrovnik, Dubrovnik, Croatia

2 Brodarski Institute, Zagreb, Croatia

3 University of Dubrovnik, Dubrovnik, Croatia

## References

[1] P.Werbos "Aproximate dynamic programming for real-time control and neural modeling" in Handbook of Intelligent Control, White and Sofge Eds. New York: Van Nostrand Reinhold, pp493-525.

[2] D. Prokhorov: "Adaptive critic design", IEEE Trans. Neural Networks, vol. 8, pp. 997-1007, Sept 1997.

[3] Wiliams, R. J. and Zisper D: "A learning agorithm for continually running fully recurrent neural networks", Neural Copmutation, 1:270-280.

[4] Pedro DeLima: "Application of Adaptive Critic Designs for Fault Tolerant Control", IEEE Computational intelligence Society Walter J Karplus Summer Research Grant 2004.

[5] R.E. Kalman, ‘‘A new approach to linear filtering and prediction problems,’’Transactions of the ASME, Ser. D, Journal of Basic Engineering, 82, 35–45 (1960).

[6] G.V. Puskorius and L. A. Feldkamp: "Neurocontrol of Nonlinear Dynamic Systems with Kalman Filter Trained Recurrent Networks", IEEE Trans. on Neural Networks, 5(14), 279-297.

[7] Simon S. Haykin: “Kalman Filtering and Neural Networks”, 2001

[8] M. Cernasky, L. Benuskova:“ Simple recurrent network trained by RTRL and extended Kalman filter algorithms“ Neural Network World 13(3) (2003) 223–234

[9] Narendra K.S., Parthasarathy K. “Identification and Control of Dynamical Systems Using Neural Networks”, IEEE transaction on Neural Networks, 1:4-27, March 1990.

[10] IEEE recommended practice for excitation system models for power system stability studies, : IEEE St. 421.5-2002 (Section 9).

[11] W. Mielczarski and A. M. Zajaczkowski "Nonlinear Field Voltage Control of a Synchronous Generator using Feedback Linearization", Automatica Vol 30, pp1625-1630, 1994.

[12] P. Shamsollahi, O. P. Malik, "Real time Implementation and Experimental Studies of a Neural Adaptive Power System Stabilizer", IEEE Trans. on Energy Conversion, Vol. 14. No. 3, September 1999.

[13] G. K. Venayagamoorthy, Ronald G. Harley, and Donald Wuncsh "Implementation off an adaptive neural network for efective control of turbogenerators” *IEEE Budapest Power Tech. Conf. 1999. paper BPT99 pp 431 – 23.*

[14] T. Kobayashi and A. Yokoyama, "An Adaptive Neuro-Control System of Synchronous Generator for Power system stabilization", IEEE Trasaction on Energy Conversion, Vol. 11, No. 3, September 1996.

[15] D.Flynn, S. McLonne, G. W. Irwin, M. D. Brown, E. Swidenbank, and B. W. Hogg, "Neural control turbogeneraroe systems", Automatica, vol 33, no,11, pp. 1961 – 1973, 1997.

[16] Q.H. Wu, B.W. Hogg, G.W. Irwin " A neural network regulator for turbogenerators," IEEE Trans. on Neural Network, vol. 3, no. 1, pp. 95-100, Jan 1992.

[17] G. K. Venayagamoorthy, Ronald G. Harley, and Donald Wuncsh "Implementation off an adaptive neural network for efective control of turbogenerators" IEEE Budapest Power Tech. Conf. 1999. paper BPT99 pp 431 – 23.

[18] MATLAB the Language of Technical Computing, http://www.mathworks.com/help/rtwin/index.html

[19] MATLAB the Language of Technical Computing, http://www.mathworks.com/products/matlab

[20] M.Miskovic " Extended Synchronous Generator Operation Region Using Neural Network Based Adaptive Control ", PhD thesis, FER Zagreb 2007.

[21] M. Miskovic, M.Mirosevic, G.Erceg "*Load angle estimation of a synchronous generator using dynamical neural networks*"//Energija 02 (2009.) ; 174-191.

[22] Miskovic, Mato; Mirosevic, Marija; Miskovic, Ivan. On-line Identification of Synchronous Generator Mathematical Model // ICRERA 2012 Proceedings / Kurokawa, Fujio (ur.). Nagasaki, Japan : University of Nagasaki, 2012. 1-3

[23] Matuško, Jadranko; Petrović, Ivan; Perić, Nedjeljko. Neural network based tire/road friction force estimation. Engineering Applications of Artificial Intelligence. (2007), 15 pages

[24] Sumina, Damir; Bulic, Neven; Miskovic, Mato. Parameter tuning of power system stabilizer using eigenvalue sensitivity. // Electric power systems research. 81 (2011), 12; 2171-2177