Synthesis of Neurocontroller for Multirotor Unmanned Aerial Vehicle Based on Neuroemulator

Sergey Andropov, Alexei Guirik
ITMO University
Saint-Petersburg, Russia
Andropov.Sergey1@gmai.com, avg@corp.ifmo.ru

Mikhail Budko, Marina Budko
ITMO University
Saint-Petersburg, Russia
{mbudko, mbbudko}@corp.ifmo.ru

***Abstract*—This paper presents a method of creating a neurocontroller based on a multilayer perceptron for an unmanned aerial vehicle. We show how a neural network can effectively emulate dynamic characteristics of an aerial craft. Another network learns to control the emulator, using backpropagation algorithm to calculate the error in its control signal. A set of parameters is used to analyze the efficiency of the stabilization and the weights of the neurocontroller are adjusted accordingly. It is shown that the system meets stabilization requirements with sufficient number of iterations. Described method can be used to remotely control unmanned aerial vehicles operating in changing environment.**

I. Introduction

An ability to solve a wide array of tasks is one of the distinct advantages of small scale multirotor unmanned aerial vehicles (UAV). Applications of UAVs range from aerial reconnaissance to warfare operations. Because of this, UAVs have to deal with different kinds of environments and, as a result, with unpredictable turbulences. Such conditions, even despite the possible geometrical simplicity of the craft, make it difficult to properly control and stabilize a flying vehicle [1], [2].

One of the most popular ways of stabilizing a multirotor UAV is a Proportional-Integral-Derivative (PID) controller. However, even with proper tuning a PID controller is unable to account for all non-linearities that influence the behavior of the craft.

A promising way to address these problems is to use artificial neural networks (ANN) [2], [3]. Replacing a traditional PID controller with a neural network makes it possible for the system to adapt to changing conditions, as well as optimize the performance of a control scheme in a way that is not achievable for a conventional controller.

One of the feasible applications of artificial neural networks is using the network to tune the coefficients of a PID controller [11]. However, such systems are inherently limited by the abilities of the controller itself. Another downside to this approach is the inability to change the target's performance parameters. For example, a quick, responsive craft requires a different set of coefficients compared to a more stable, but slower one. Usually, these qualities are only empirically described.

These and other problems can be solved by using artificial neural networks. Generally, neurocontrol is used in two distinct ways: direct and indirect. Direct control includes inverse neural emulators, predictive control techniques and adaptive critics [4], [5], [6]. Indirect control involves hybrid methods and parallel neurocontrol [7].

We propose a method of emulating the object and using it to teach an adaptive neurocontroller, which satisfies a set of requirements such as overshoot, settling time, steady-state error and rise time. The results show an increase in the quality of control and reduction of the amplitude of damped oscillations compared to a conventional PID controller.

The remainder of this paper is structured as follows. Section 2 describes the aerial vehicle and the control loop. Section 3 presents the design of neuroemulator, learning algorithm and neurocontroller application scheme. In section 4 we present the results of training experiments.

II. Control loop and stabilization object

PID control is the most common control algorithm used in industry. It is applicable to a wide range of operating conditions.

A control system functions in a loop, in which it acquires sensor data and calculates the error between current state and a set point. Based on that error it computes control signal and applies it to the system. This type of control system is known as a closed-loop system, or a feedback control system. A PID controller takes sensor data and computes the desired actuator output by calculating proportional, integral, and derivative responses and summing those three components to compute the output. All computations required are linear, can be optimized and do not take significant processing power.

We experimented with a small four rotor UAV of our own design [1]. Its layout is shown in Fig. 1.

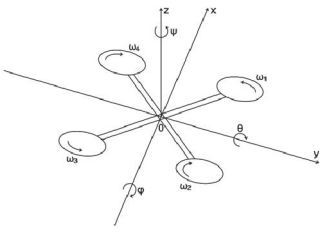

Fig. 1. Quadcopter layout

This aerial craft uses a gyroscope, accelerometer and magnetometer to determine its position in a 3 dimensional space. It does so by computing roll, pitch and yaw angles. The movement of quadcopter is controlled by changing the rotation speed of rotors.

Stabilizing an aerial vehicle with PID controller requires several variables – current angle error (difference between an actual and a desired tilt angle of UAV), angular speed and accumulated error. Each variable is processed with its own coefficient and has a different impact on the controller's output. Controller that is used for stabilization by roll angle (roll controller) is identical to pitch controller and they often are configured simultaneously and share coefficients, however, coefficients for yaw controller generally differ.

Tuning a PID controller involves changing coefficients of proportional, integral and derivative parts of the controller. There are multiple methods of tuning the PID-controller coefficients [10]. However, a number of these requires a precise mathematical model of the control object; methods that do not use a model have their own disadvantages, such as still needing a manual tuning after the algorithm adjusted the initial values. Thus, PID tuning is often done manually by changing the coefficient's value for each component and observing the UAV's response to the change.

Without any external disturbance, a well tuned PID-controller can stabilize a UAV while avoiding using excessive resources. However, in a real world scenario the system needs to be more agile and able to adapt to ever changing conditions. Any change in the external (temperature, wind, atmospheric pressure), or internal (weight, geometry) environment might cause a set of coefficients to be sub-optimal. Additional tuning will be required, which is difficult to do on the fly.

III. Neural network architecture

An artificial neural network is a computational system inspired by the way biological organisms, such as the human brain, process information. A neural network does not follow a linear path; instead, any input data is processed in parallel through a set of nodes. These nodes are called “neurons”.

The most important aspect of a neural network is its ability to adjust the way it processes the information through a learning algorithm, thus changing the output. This is achieved through modification of “weights” (which regulate the impact of each connection on the final result) and “biases”.

It is known that neural networks are effective in approximating an unknown non-linear function, if some information about the function is provided [3]. They are capable of finding patterns and trends within complicated or imprecise data.

There are several ways of incorporating ANNs in neural control: linear neurocontrollers, multilayer perceptrons, recurrent neural networks [3], [4], [7].

Their qualities make it possible to use neural networks in different ways:

- pattern recognition;
- adaptive control;
- object emulation;
- anomaly detection;
- time series prediction.

In this work we use several multilayer perceptrons for two distinct goals:

1) Emulating the control object
2) Creating a neurocontroller

A. Basic neural network structure

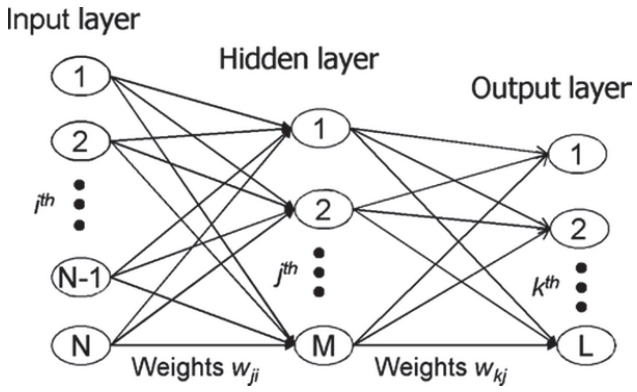

Fig. 2. Neural network scheme

Neural networks used in this work are feed-forward networks, which means the signal travels in one direction – from input to output. The output of any layer does not affect that same layer.

Networks consist of several layers: input, output and one or several “hidden” layers.

The input layer represents the raw information that is fed into the network.

Second, "hidden", layer allows network's behavior to be non-linear. The number of neurons within that layer and the number of hidden layers itself can be chosen with different techniques in mind and depends on the conditions of the task, however, for relatively simple tasks one layer is usually enough [9]. The number of neurons can be calculated as an average between input and output neurons, but might be increased.

Third layer is an output layer with one or several output neurons, as per the task upon the network.

Each neuron in the hidden layer has an activation function for producing non-linear output and propagating it forwards through the network. We experimented with a number of activation functions and found most activation functions producing similar results in the rate of convergence and computational load with small differences in the derivative calculation and storage.

A sigmoid function was chosen:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

where x is a sum of inputs xi with weights ωi and bias b:

$$x = \sum \omega_{ij} x_i + b$$

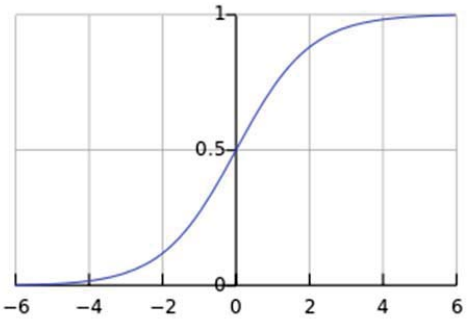

Fig. 3. Sigmoid function

This function is bounded, easily differentiable, monotonic, and produces a smooth output (Fig. 2). Small changes in input coefficients (weights and bias) result in small changes in the output of the function. Its output lies in [0;1] range, which makes it useful in classification cases.

Another great quality of this function is that its derivative:

$$o'(x) = o(x)(1 - o(x))$$

can be easily computed since we can save the result of the function while propagating the input signal through the network. The derivative of the sigmoid function is used later in backpropagation, and storing it reduces processing time.

For the output layer we use a linear activation function, since the output of the network lies outside of the sigmoid's function range. A linear activation function is equal to its input. As a result, it is not bound and can produce any value.

B. Network learning algorithm

One of the most common training algorithms for artificial neural networks is backpropagation [8]. This algorithm can optimize weights and biases in a multilayer network.

The backpropagation algorithm uses gradient descent method to look for the minimum of the error function. A solution is thus the combination of weights that minimizes the error.

First, the input information is presented to the network and propagated forward until it reaches the output layer. Then the desired and actual outputs are compared and the error for each output neuron is calculated. This error is propagated backward through the network, thus giving the error for each neuron in all hidden layers. Using these values, a backpropagation algorithm can update weights and biases.

In order to use this method, we need a training set consisting of n ordered pairs of multidimensional vectors $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, in other words – input and output patterns. This dataset maps several inputs to outputs, establishing a pattern for the neural network to learn.

Initial weights ω_{ij} and ω_{jk} of the network are selected at random. When input xi is presented to the network, it is propagated through the network, producing an output o_i. The goal of the training algorithm is to make the output o_i close or identical to the desired output t_i for each input. This is done by minimizing the error function:

$$E = \frac{1}{2}\sum_{i=1}^{n}(o_i - t_i)^2$$

First, the error signal in the output layer k is calculated:

$$\Delta_k = t_k - o_k$$

$$\delta_k = \Delta_k o'_k(x) \quad (1)$$

where $o'_k(x)$ is a derivative of the activation function. For the output layer this derivative equals 1.

The weights of the output layer are adjusted according to:

$$\Delta\omega_{jk} = x_k \delta_k \gamma \quad (2)$$

where x_k is the input from a neuron in the previous layer (i.e. the output of the relative neuron in the hidden layer), γ is the learning rate. This learning rate is typically a small number (eg. 0.004), regulating the speed at which the weights are adjusted. Higher learning rate values may cause the network's outputs to oscillate around the target, thus never converging on a solution; small values might cause the learning process to be very slow.

It is worth noting that the gradient descend method has a downside in that it might get "stuck" at the local minimum of the error function. In order to get over the "small hill" and continue moving toward a global minimum, we can modify the equation (2) as follows:

$$\Delta\omega_{jk}{}^n = x_k \delta_k \gamma + \Delta\omega_{jk}{}^{n-1}\varphi$$

where φ is a *momentum factor*. The introduction of the momentum accelerates the learning process by keeping track of the previous changes, thus allowing the algorithm to move in larger steps. The faster movement prevents the network from settling in a local minimum by helping it move past the "hill".

The error signal for the nodes in hidden layer is calculated in a similar way to the output layer.

$$\delta_j = o'_j \sum \omega_{jk}\delta_j$$

where $\sum \omega_{jk}\delta_j$ is the weighted error signal. A derivative of the activation function for the hidden layer is:

$$o'_j = o_j(1 - o_j)$$

therefore

$$\delta_j = o_j(1 - o_j)\sum \omega_{jk}\delta_j \quad (3)$$

Weights of the hidden layer are updated in the same way as the weights of the output layer:

$$\Delta\omega_{ij}{}^n = x_j \delta_j \gamma + \Delta\omega_{ij}{}^{n-1}\varphi$$

Biases of both layers are updated in a similar way as weights:

$$\Delta b_{ij} = \delta_j \gamma$$

$$\Delta b_{jk} = \delta_k \gamma$$

C. UAV emulation

Before training our neurocontroller, we first create a neural network that is able to behave like the target object.

Training a neural emulator of the object involves collecting a dataset of object's states, inputs and corresponding outputs. A neural network that has the same number of inputs and outputs is created, with one hidden layer and a configurable number of neurons. As mentioned before, the number of hidden neurons depends on the particular task and could be determined empirically. After experimenting with different number of neurons we chose 11 nodes in the hidden layer, as this particular number allowed the network to converge on the solution relatively fast and with acceptable precision.

The network has 3 inputs – a control signal, current state, which in our case is the tilt angle, and current speed. After processing the inputs, the emulator returns the resulting state and speed of the system.

An important note here is the fact that we use the same model for roll and pitch angles, since the dynamics of these angles in the real UAV are nearly identical and often the same set of PID coefficients is used for controllers of both angles. A separate model for the yaw angle should be used. The method of training the neural network to emulate the UAV on yaw angle is identical to the one used for roll/pitch angles, therefore in the scope of this article we will only present experiment results for the latter.

Training is done in the following way: we collect flight data from the real quadcopter and separate it into the training set and the testing set. Using backpropagation algorithm, we train the neural network to match the outputs of the real craft. After multiple iterations and when the squared mean error (4) is below the acceptable threshold, the model is tested with the values that were not present in the training set to determine how well the emulator represents the target object.

$$E = \frac{\sum_{i=1}^{n}(y_{ni} - y_i)^2}{n} \quad (4)$$

The process is represented in Fig. 4.

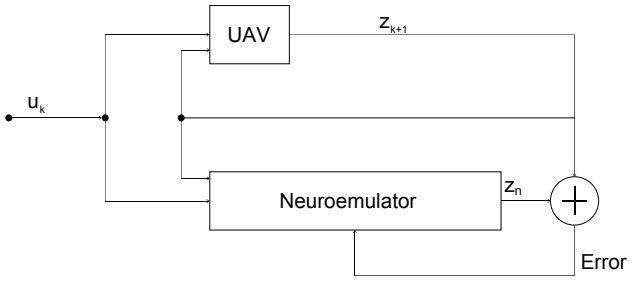

Fig. 4. Neural network UAV emulation

The initial state Z_0 of the UAV is passed to the neural network along with the PID control signal. The network produces a new state Z_n, which is then compared to the actual state Z_{k+1}; the error is backpropagated through the network and its weights and biases are adjusted. The process repeats until the conditions of squared mean error being below the target mark are met.

D. Neurocontroller initial training

The goal of this work is to create a controller capable of adapting to the changing conditions, that satisfies a set of requirements. Using a neural network makes adjusting the neurocontroller attitude possible; this controller is also not bound by the limitations of the PID algorithm.

In order to achieve a better stabilization response, we increase the number of inputs that the controller takes. Aside from the normal three inputs (current error of the angle, angular speed and accumulated error), the neurocontroller also takes acceleration and the desired rise time as input. The output of the controller is the same output a PID algorithm gives.

Having a neural emulator of the target object means we can train the network without the risk of breaking the actual vehicle. The learning process always starts with the object being in a state z_0; the goal of the controller is to drive the vehicle to the desired state z_d. While training, we set the initial tilt angle of the model to 30 degrees on one of the axis with 0 angular speed. The desired state is set to 0 degrees and 0 angular speed. This means that after a period of t the controller needs to drive the model to the desired state and stop there. Any difference in the angle or the speed contributes to the total error of the controller.

A controller stays the same within one stabilization cycle for a period of t. Since initially it does not possess any knowledge about how to control the model, we use a conventional PID controller data to quickly train the neurocontroller to stabilize the model just like PID algorithm would. Thus, our neurocontroller training is done in two phases. In the first phase it learns to behave identically to a PID controller. Inputs not supported by PID algorithm are set to zero in this phase.

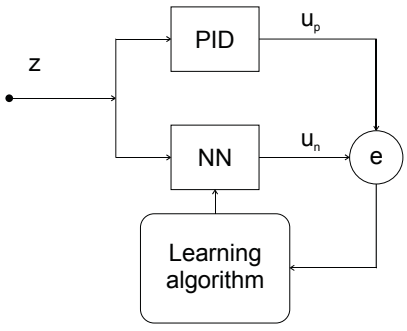

Fig. 5. Neurocontroller initial training

A learning scheme is presented in Fig. 5. Since the PID function has linear relation to its arguments, the network converges on the solution relatively fast. After a number of iterations the neurocontroller is capable of stabilizing the craft on a level of a well tuned PID controller and is now ready to be improved.

E. *Neurocontroller tuning*

A trained controller can be tuned further, using the neuroemulator to test its performance. This is the second phase of controller training.

Our goal is to modify the weights and biases according to the final state of the model. To train the controller, we need to know the error of the control signal; unfortunately, the only available error is the difference between final state and the desired state. However, since the vehicle model is a neural network, we can backpropagate the error of the output layer all the way back to the input layer, which means the controller's error can be acquired. The real vehicle cannot be used in this way, because there is no data on the relation between its inputs and outputs.

Using the equations (1) and (3), we translate the final state error to the neurocontroller output error. Figure 6 represents the training process. This method was suggested by Derrick H. and Bernard W. [3] and is a very effective tool to bypass the usual limitation of the backpropagation algorithm – obtaining a proper labeled dataset. Since normally there is no particular data on how *exactly* the object should be moving at each point in time, training the neurocontroller is difficult as we do not have the information required to compute the error signal for all layers. However, with this approach a controller will learn the optimal way to get to the target by itself, using the data from the emulated vehicle to acquire its own error.

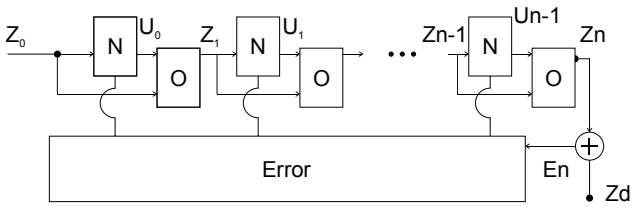

Fig. 6. Neurocontroller training with neuroemulator

IV. Results

Neurocontroller showed promising results in learning to control the emulated model of quadcopter. As mentioned earlier, the training was done in two phases: first, we created functional PID analog; second, using algorithm described in section 3, we tuned the neurocontroller further. A set of requirements was chosen for the network to meet: overshoot less than 10% of the initial angle, rise time of 1 second, settling time of 1.5 seconds and the steady-state error of 0.1%. Final test involved stabilization in changing external conditions, which were generated artificially by applying a random changing force to the craft model.

After the first phase, the neurocontroller behaved nearly identically to conventional PID, which is shown in Fig. 7.

In the second phase the network started to receive the error, propagated backwards through the neuroemulator, and adjusts its weights accordingly. Initially after 1000 iterations the network appeared to give worse results (Fig. 8), however after 100000 and 200000 of iterations it showed visible improvement (Fig. 9 and 10).

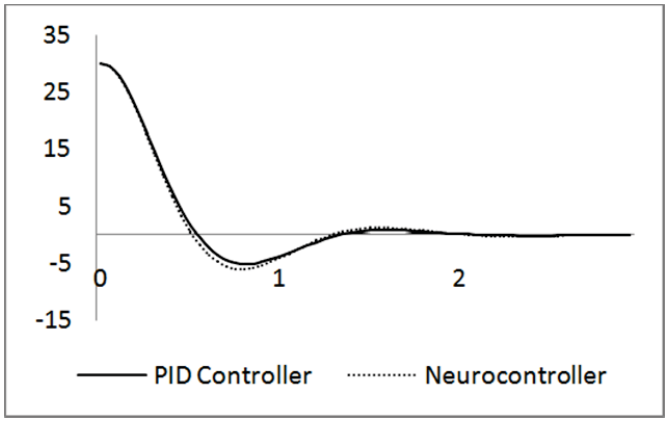

Fig. 7. PID and neurocontroller comparison

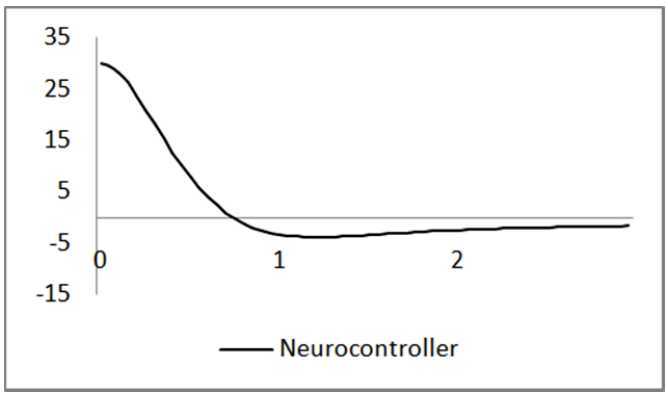

Fig. 8. Neurocontroller stabilization after 1000 learning iterations

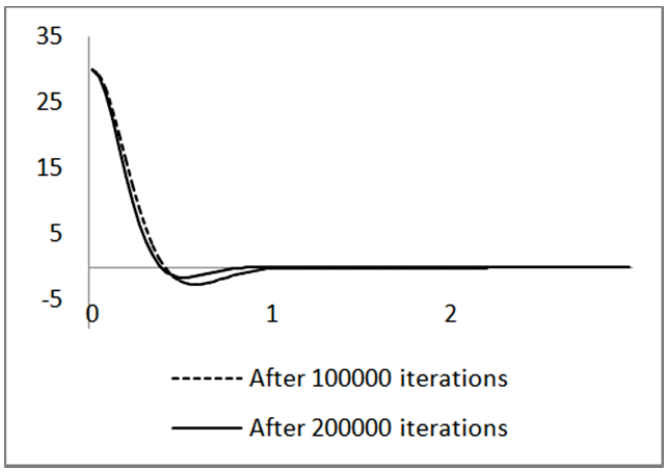

Fig. 9. Neurocontroller stabilization after 100000 and 200000 learning iterations

It is apparent that introduction of new inputs throws the neurocontroller off initially. Nonetheless, with small learning rates the neural network is capable of adapting to the new conditions.

It is important to note that, while the number of iterations required for the training process may seem large, the actual computational load in flight is significantly lower, since the

controller is already tuned and only needs minute adjustments. Most of the initial training for the neurocontroller can be done on the emulated object, and the emulation training process only requires flight data from the real UAV.

Flight tests were done using the ARM32 Cortex-M3 80Mhz processor. It proved more than capable of handling 2 neural networks (for yaw and pitch/roll stabilization), as well as other functions required for remote control.

It can therefore be concluded that the presented method of using a neurocontroller based on a multilayer perceptron is an effective way to build a flight controller for a multirotor UAV.

Future work will be focused on the optimization of the learning algorithm and reduction of time required for network training.

V. Conclusions

Mechanical simplicity of UAVs comes at a cost of increased controller complexity. Quadrotors, unlike certain other aircrafts, are inherently unstable and highly sensitive to small changes in rotor speeds. In this paper, we presented a controller based on artificial neural networks. Initially the neurocontroller learns to control the aircraft similarly to PID algorithm. The neuroemulator is used to further tune the controller, thus achieving better results, especially in the presence of external turbulences.

Acknowledgments

This work was supported by RSF grant 16-11-00049.

References

[1] Belyaev S.S., Budko M.B., Budko M.Y., Guirik A.V., Zhigulin G.P. "Functional design of flight and navigation controller unit for multirotor unmanned aerial vehicle", *Radio Industry,* vol. 4, 2015, pp. 77 – 87.

[2] Bobtsov A.A., Guirik A.V., Budko M.Yu., Budko M.B. "Hybrid Parallel Neuro-controller for Multirotor Unmanned Aerial Vehicle", *Proceedings of 8th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops* (ICUMT), 2016 – pp. 32-35.

[3] Derrick H. N., Bernard W. "Neural Networks for Self-Learning Control Systems", *IEEE Control Systems Magazine*, Apr. 1990.

[4] Lendaris G.G. "A Retrospective on Adaptive Dynamic Programming for Control", *Proceedings of International Joint Conference on Neural Networks*, Atlanta, USA, June 14-19, 2009, pp. 1750 – 1757.

[5] Narendra K.S., Parthasarathy K.K. "Identification and control of dynamical systems using neural networks", *IEEE Transactions on Neural Networks,* 1990, vol. 1, pp. 4 – 27.

[6] Borisov O.I., Gromov V.S., Pyrkin A.A., Bobtsov A.A., Nikolaev N.A."Output Robust Control with Anti-Windup Compensation for Quadcopters", *IFAC-PapersOnLine*, vol. 49, Issue 13, pp. 287–292.

[7] Hagan M.T., Demuth H.B. "Neural networks for control", *Proceedings of the American Control Conference*, San Diego, USA, 1999, vol. 3, pp. 1642 – 1656.

[8] Rojas R. *Neural Networks. A Systematic Introduction.* Berlin: Springer-Verlag, 1996.

[9] Karsoliya S. "Approximating number of hidden layer neurons in multiple hidden layer BPNN architecture", International Journal of Engineering Trends and Technology, 2012, vol. 31, N6., pp. 714–717.Rojas R. *Neural Networks. A Systematic Introduction.* Berlin: Springer-Verlag, 1996.

[10] Ziegler J.G., Nichols N.B. "Optimum settings for automatic controllers", *Trans. ASME*, 1942, vol. 64, pp. 759 – 768

[11] Evgenov A.A. "Neuro-controller of quadcopter control system", *Modern Problems of Science and Education*, 2013, no. 5, p. 61.