Chapter 8

Model Reference Adaptive Control of Quadrotor UAVs: A Neural Network Perspective

Nikhil Angad Bakshi

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/intechopen.71487

Abstract

Adaptive models and online learning are two equivalent topics under different umbrellas of research – control systems and machine learning. This chapter will tackle one such application of a neural network-based model reference adaptive controller on a quadrotor unmanned aerial vehicle while stating the general principles behind each design decision so the knowledge can be generalized to other practical applications. The application-oriented presentation of this chapter will run parallel to most research and development processes in the field, where the physical system or a simulator is usually available and a simple control system (such as PID) has already been implemented as a baseline. The black-box nature of a neural network can truly be leveraged to improve performance after reading this chapter. Several practical considerations when approaching such a problem have been discussed together with their general and implemented solutions. The simulation results for the problem have been presented to demonstrate the success of this control strategy.

Keywords: model reference adaptive control, neural networks, UAV, quadrotor, machine learning, robotics, online learning, MLP networks

1. Introduction

Artificial intelligence is a term that, very paradoxically, holds a prerequisite for the absence of intelligence, which the designer must then overcome. Intelligence can be viewed as the ability of a system to *perceive* its environment, *reason* upon the acquired *knowledge* and perform an *action* or task based on this information to meet its *objective*.

When the possible states of the environment are predictable the designer can create an intelligent system that *performs* well for all possible situations. However the world is a messy place

and more often than not the environment is unpredictable and knowing or encoding such information into the system a priori is impractical.

The set of *actions* of a system and its *objective,* on the other hand, is usually known a priori, so it logically follows that one should design a system that should be able to *learn* how to deal with new situations to meet its *objective* given the limited set of *actions.*

Formally, the system or agent should improve its performance as measured by a metric (P) on a task (T) with increasing experience (E). This brings us to *machine learning* (ML).

At this point, let us note that an *adaptable control system* is one that modifies the *control law* so that the system remains stable and the *control objective* is met.

Whether one looks at it from the perspective of ML, in that the system is initially poor at meeting the objective and hence it changes system parameters to improve or from the control theory perspective that the environment or system has changed and the control objective is not met demanding a change in the control law, we are describing a similar situation.

This chapter is written using the *attitude* and *altitude* controller of a quadrotor unmanned aerial vehicle (UAV) as a running example however every idea will be presented generally at first and then tied back to the practical example in consideration.

2. Quadrotor system

A quadrotor UAV (refer **Figure 1**) has four motors that independently provide thrust (as indicated by F_1, F_2, F_3 *and* F_4) and based on these thrusts the UAV can change its *attitude* (roll ϕ, pitch θ and yaw ψ) and *altitude* (z). This chapter will focus on the inner loop control of the *attitude* and *altitude* variables and is based on the work of Bakshi and Ramachandran [1].

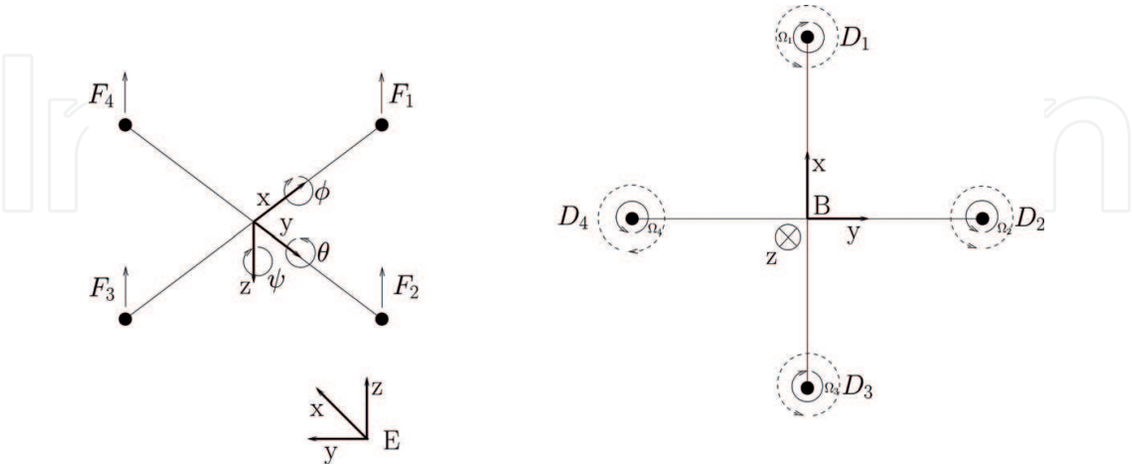

Figure 1. Quadrotor coordinate system [2].

The horizontal coordinates of the UAV (x, y) can be controlled using an outer loop on this inner loop but that is not covered here, we shall assume that the environment of the quadrotor is boundless in all directions except the datum in z, i.e. the ground.

The quadrotor model that has been used is based on work done by Bouabdallah et al. [2]. Following Bouabdallah, the earth-fixed frame E and the body-fixed frame B are as seen in **Figure 1**. The aerodynamic forces and moments considered in the model are based on the work of Gary Fay as in Ref. [3].

A dynamic model for the quadrotor is used for the purpose of simulation in this chapter so that the control strategy that has been presented here can be thoroughly evaluated.

2.1. Quadrotor parameters

The quadrotor parameters are based on the work of Bouabdallah [4] (**Table 1**).

Design Variable	Value	Units	Description
m	0.53	kg	Mass
L	0.23	m	Craft diameter
J_r	6×10^{-5}	$kg.m^2$	Rotor inertia
I_{xx}	6.23×10^{-3}	$kg.m^2$	Moment of inertia along x
I_{yy}	6.23×10^{-3}	$kg.m^2$	Moment of inertia along y
I_{zz}	1.12×10^{-2}	$kg.m^2$	Moment of inertia along z
b	3.13×10^{-5}	$N.s^2$	Thrust factor in hover
d	7.50×10^{-5}	$N.s^2$	Drag factor in hover
N	2	—	Number of blades
R	0.15	m	Propeller radius
c	0.04	m	Chord
θ_0	0.26	rad	Pitch of incidence
θ_{tw}	0.05	rad	Twist pitch
a	5.70	—	Lift slope
C_d	0.05	—	Airfoil drag coefficient
A_c	0.01	m^2	Helicopter center hub area
ρ	1.29	kg/m^3	Air density
ϑ	1.80×10^{-5}	$Pa.s$	Air viscosity
V	3.04×10^{-4}	m^3	Volume

Table 1. Quadrotor parameters [4].

2.2. Moments and forces on the quadrotor system

See **Table 2**.

Rolling moments	Body gyro effect	$\dot{\theta}\dot{\psi}\left(I_{yy}-I_{zz}\right)$
	Propeller gyro effect	$J_r\dot{\theta}\Omega_r$
	Roll actuators action	$l(-T_2+T_4)$
	Hub moment due to sideward flight	$h\left(\sum_{i=1}^{4}H_{yi}\right)$
	Rolling moment due to forward flight	$(-1)^{i+1}\sum_{i=1}^{4}R_{mxi}$
Pitching moments	Body gyro effect	$\dot{\varphi}\dot{\psi}(I_{zz}-I_{xx})$
	Propeller gyro effect	$J_r\dot{\varphi}\Omega_r$
	Pitch actuators action	$l(T_1-T_3)$
	Hub moment due to forward flight	$h\left(\sum_{i=1}^{4}H_{xi}\right)$
	Rolling moment due to sideward flight	$(-1)^{i+1}\sum_{i=1}^{4}R_{myi}$
Yawing moments	Body gyro effect	$\dot{\varphi}\dot{\theta}\left(I_{xx}-I_{yy}\right)$
	Inertial counter-torque	$J_r\dot{\Omega}_r$
	Counter-torque unbalance	$(-1)^{i}\sum_{i=1}^{4}Q_i$
	Hub force unbalance in forward flight	$l(H_{x2}-H_{x4})$
	Hub force unbalance in sideward flight	$l\left(-H_{y1}+H_{y3}\right)$
Forces along z axis	Actuators action	$c\psi c\varphi\sum_{i=1}^{4}T_i$
	Weight	mg
Forces along x axis	Actuators action	$c\psi c\varphi\sum_{i=1}^{4}T_i$
	Hub force in x axis	$-\sum_{i=1}^{4}H_{xi}$
	Friction	$\frac{1}{2}C_xA_c\rho\dot{x}\lvert\dot{x}\rvert$
Forces along y axis	Actuators action	$(-c\psi s\varphi+s\psi s\theta c\varphi)\left(\sum_{i=1}^{4}T_i\right)$
	Hub force in y axis	$-\sum_{i=1}^{4}H_{yi}$
	Friction	$\frac{1}{2}C_yA_c\rho\dot{y}\lvert\dot{y}\rvert$

Table 2. Quadrotor moments and forces summary [4].

2.3. Equations of motion

$$I_{xx}\ddot{\varphi}-\dot{\theta}\dot{\psi}\left(I_{yy}-I_{zz}\right)+J_r\dot{\theta}\Omega_r+l(-T_2+T_4)-h\left(\sum_{i=1}^{4}H_{yi}\right)+(-1)^{i+1}\sum_{i=1}^{4}R_{mxi} \tag{1}$$

$$I_{yy}\ddot{\theta} - \dot{\varphi}\dot{\psi}\,(I_{zz} - I_{xx}) - J_r\dot{\varphi}\Omega_r + l(T_1 - T_3) - h\left(\sum_{i=1}^{4} H_{xi}\right) + (-1)^{i+1}\sum_{i=1}^{4} R_{myi} \tag{2}$$

$$I_{zz}\ddot{\psi} - \dot{\theta}\dot{\varphi}\left(I_{xx} - I_{yy}\right) + J_r\dot{\Omega}_r + (-1)^{i}\left(\sum_{i=1}^{4} Q_i\right) + l(H_{x2} - H_{x4}) + \; l\left(-H_{y1} + H_{y3}\right) \tag{3}$$

$$m\ddot{z} - mg - (c\psi c\varphi)\sum_{i=1}^{4} T_i \tag{4}$$

$$m\ddot{x} - (s\psi s\varphi + c\psi s\theta c\varphi)\sum_{i=1}^{4} T_i - \sum_{i=1}^{4} H_{xi} - \frac{1}{2}C_x A_c \rho \dot{x}|\dot{x}| \tag{5}$$

$$m\ddot{y} - (-c\psi s\varphi + s\psi s\theta c\varphi)\sum_{i=1}^{4} T_i - \sum_{i=1}^{4} H_{yi} - \frac{1}{2}C_y A_c \rho \dot{y}|\dot{y}| \tag{6}$$

2.4. Rotor dynamics

A first order closed loop transfer function is used to reproduce the dynamics between the propeller's speed set point and its true speed as in Ref. [2].

$$G(s) = \frac{0.936}{0.178s + 1} \tag{7}$$

2.5. Summary of model

The following is a breakdown of the basic understanding of a quadrotor UAV system required for the purposes of this chapter:

a. The sum of the four thrusts (as indicated by F_1, F_2, F_3 *and* F_4 in **Figure 1**) along the z axis is responsible for countering the weight of the craft. Any surplus or shortage of the total thrust along z will result in motion in the z-direction.

b. An imbalance in the forces F_2 *and* F_4 will result in a rolling motion along the x-axis, similarly imbalance in F_1 *and* F_3 will result in a pitching motion along the y-axis. Note that the very act of rolling or pitching tilts the craft such that the motor thrusts are no longer effected purely in the z-direction, causing the craft to descend, therefore the total thrust must be summarily increased so that the component along z is maintained.

c. The rotations of motors 1 and 3 are in the same direction and the opposite of motors 2 and 4. To achieve a yawing motion increased thrust must be applied to a diametrically opposite pair (say 1 and 3) and reduced thrust on the other pair (2 and 4).

d. Various second order effects come into play, which have been modeled but understanding them is not crucial to this chapter.

e. This model dynamically calculates the thrust and drag coefficients which results in increased accuracy with the real world scenario.

3. Objective

The basic inner loop controller of any helicopter deals with maintaining a specified height above ground, i.e. *altitude*, and maintaining a particular pose, or *attitude*. The *attitude* in turn allows the helicopter to translate in the *x*-*y* plane assuming *altitude* is held constant.

The standard approach is decentralized or cascaded PID controllers for the various control variables (in this case: roll ϕ, pitch θ, yaw ψ, altitude z), these controllers will have to be tuned for each particular quadrotor UAV. In general, any non-adaptive controller will need to be tuned to a particular quadrotor.

In this chapter we employ neural networks to design an adaptive controller that is system unspecific, i.e. it should work for any quadrotor system. It learns the system parameters online, i.e. in-flight. The challenge is to keep the system stable during online learning.

4. Indirect model reference adaptive control

Indirect adaptive control is when the controller estimates the plant to predict its state variables and these are used to modify the controller parameters.

Direct adaptive control is when there is no plant identification, instead the controller parameters are modified on the basis of the error the system has with the reference.

MRAC is a direct adaptive control method (refer **Figure 2**), however in this chapter we shall be taking a mixed approach to the problem.

In MRAC we define a reference model that responds to the input signal (r) as we would like our plant to respond. The controller generates a control signal (u) based on its control law which it expects will make the plant output (y) follow the reference output (y_{ref}). Depending on the deviation (or error), the adjustment mechanism will update the control law (by modifying parameters) of the controller. This process is repeated iteratively so that the plant follows the reference.

The beauty of the approach taken in this chapter is that we needn't formalize the control logic. We will delve deeper into neural networks before returning to the problem at hand. For the

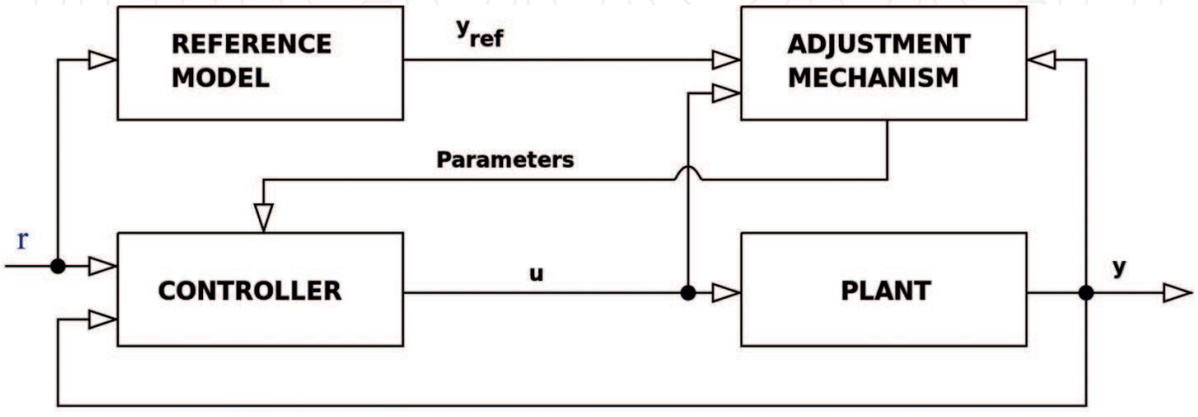

Figure 2. MRAC block diagram. Image courtesy of Wikipedia under CC license.

time being we will leave the 'reference model, 'adjustment mechanism' and the 'controller' as black boxes, they will be revisited in Section 6.

5. Neural networks

5.1. Introduction to machine learning

A formal definition of ML is:

> *A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.*
> *-Tom Mitchell*[1] (*1998*)

Neural networks are one such machine learning algorithm. This sub-section will briefly cover the two broad categories of machine learning algorithms. Bear in mind that this chapter will elaborate on neural networks used in a *supervised learning* setting.

5.1.1. Supervised learning

In supervised learning, the data is tagged with the correct values for prediction/classification associated to it. The algorithm learns by minimizing the error between its results and the correct results. This is the most common form of machine learning and the easiest, however labeled data is not easy to come by as its curation and tagging is usually expensive.

Neural networks, support vector machines (SVMs), linear/logistic regression and decision trees are a few examples of supervised learning algorithms and the applications could be classification of images, weather prediction, sentiment detection, face recognition, etc.

5.1.2. Unsupervised learning

In unsupervised learning, the task is to find patterns or meaning in untagged data such as classifying similar data together without actually knowing what those classes may represent (clustering) or we take data in some low level/uncompressed representation and learn a high level/compressed representation with minimum information loss or we have a lot of data which mostly subscribes to a particular pattern and we would like to detect the outliers (anomaly detection).

K-means clustering, autoencoders (NN based) and principal component analysis are a few algorithms used for unsupervised learning tasks.

5.2. History and intuition of neural networks

In the twentieth century scientists were able to definitively identify the primary unit of the brain – the neuron. One theory of the time was that information is not pre-loaded in the brain of a newly born child, only the basic structure and connections between the neurons in its brain exist, the brain learns to function by strengthening/weakening various neural pathways.

[1]Machine Learning, Tom Mitchell, McGraw Hill, 1997.

Therefore it was theorized that the neuron, which either fires or does not, can be modeled as a function that has multiple inputs, a single output (which may be connected to several other neurons) and only 'fires' when the sum of inputs exceeds a certain threshold. The connections between neurons have weights, which strengthen or weaken a connection between two neurons.

"Neurons that wire together, fire together"
-Donald Hebb (1949)

The above quotation may be familiar. It is based on Donald Hebb's theory to explain neural adaptation and learning and this forms the basis of learning in modern-day artificial neural networks.

5.3. Formalization

As shown in **Figure 3**, a basic neural network consists of layers of nodes (or neurons) where each node has a connection to all the nodes of the next layer, and takes input from each of the nodes in the previous layer. Each of the connections has a real number weight associated with it. Every neuron does some simple computation (we will restrict ourselves to the sigmoid activation and linear activation) on the sum of its inputs to yield an output value.

The above definition is that of multilayer perceptron (MLP) network which is the most basic form of a feed forward neural network.

Let us assume that $\boldsymbol{x}$ is the input (vector) for this neural network, the dimension of $\boldsymbol{x}$ is 6×1. Let the weight of the connections between the input layer and the first hidden layer be represented by the matrix $\theta^{(1)}$, each value in the matrix will be referenced as $\theta_{ij}^{(1)}$. The dimension of $\theta^{(1)}$ is 6×4.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{8}$$

Eq. (8) shows us the sigmoid activation function that will be applied on every node in the hidden layers. If z is a vector then applying the sigmoid activation function will result in a

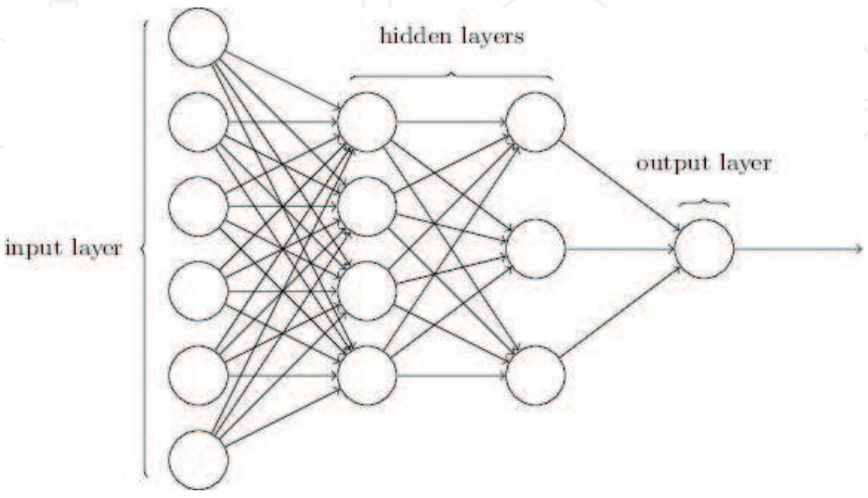

Figure 3. A depiction of a basic neural network. Image courtesy of Michael A. Nielson. Neural networks and deep learning. Determination Press, 2015.

vector of the same dimension. Sigmoid function is a continuous differentiable function, which is bounded between 0 and 1.

$$\mathbf{a}^{(1)} = \sigma\left(\left(\theta^{(1)}\right)^T \boldsymbol{x} + \boldsymbol{b}^{(1)}\right) \tag{9}$$

Eq. (9) shows us the first step in *forward propagation.* $\mathbf{a}^{(1)}$ is known as the activation of the first hidden layer. As a sanity-check we can see the dimension of $\mathbf{a}^{(1)}$ is 4×1, which is congruent with what we expect.

Neural networks have a bias unit (not shown in **Figure 3**), which is a neuron that is always firing and is connected to every node in the next layer but does not take input from the previous layer. Mathematically it can be represented as the additive term $\boldsymbol{b}^{(1)}$ of dimension 4×1 shown in Eq. (9).

$$\mathbf{a}^{(2)} = \sigma\left(\left(\theta^{(2)}\right)^T \mathbf{a}^{(1)} + \boldsymbol{b}^{(2)}\right) \tag{10}$$

$$\boldsymbol{y}_{predicted} = \boldsymbol{a}^{(3)} = \mathbf{z}^{(3)} = \left(\theta^{(3)}\right)^T \mathbf{a}^{(2)} + \boldsymbol{b}^{(3)} \tag{11}$$

Eqs. (10) and (11) complete the forward propagation. Eq. (11) can include a sigmoid activation too if the desired output is between 0 and 1 like in a classification problem. No activation function is also known as linear activation.

The neural network learns by optimizing an objective function (or cost function) such as the squared-error cost function for dealing with regression problems as in this text.

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(\boldsymbol{y}_{predicted}{}^{(i)} - \boldsymbol{y}^{(i)}\right)^2 \tag{12}$$

where $\boldsymbol{y}^{(i)}$ is vector of the target values of output layer. In our example it is 1×1 but in general the neural network can have multiple outputs. In an offline setting we have all our data beforehand, indicated here by m examples and we compute cost iterated over all $\boldsymbol{y}^{(i)}$.

As seen in **Figure 4** the sigmoid function is approximately linear between -2 and 2 and is almost a flat line beyond -4 and 4. This has 3 implications:

1. It gives us a convenient differentiable function that is nearly binary
2. If the input of a neuron is too extreme the neuron becomes saturated and therefore information gets attenuated going through the network if it is too deep.
3. In the case of saturated nodes, backpropagating, which involves computing the gradient, becomes ineffective, as the gradient at the extremes is nearly zero.

Due to the latter two points it is beneficial to ensure that the weights of the network are small. If the weights have large values then the network is sure those connections are very important which makes it hard for it to learn otherwise. Therefore, it makes sense to keep weights small so the network is responsive to new information. To accomplish this we incorporate the

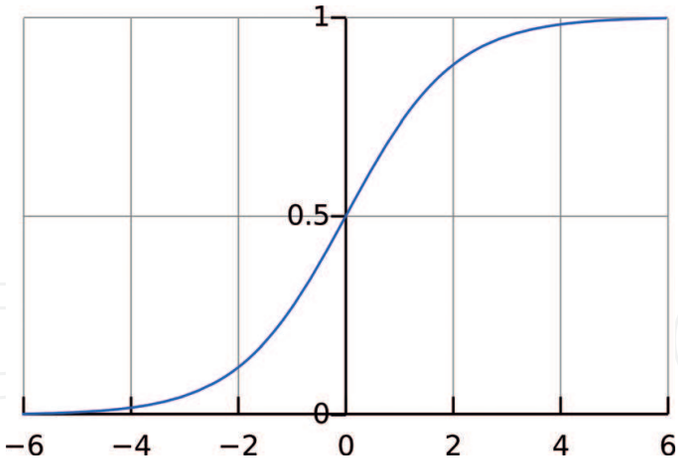

Figure 4. Graph of sigmoid function.

weights into the cost function. In L1 *regularization* we add the modulus of the weights to the cost function in Eq. (12). In L2 regularization we add the squares of the weights to the cost function resulting in Eq. (13).

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(\boldsymbol{y}_{predicted}{}^{(i)} - \boldsymbol{y}^{(i)}\right)^2 + \frac{\lambda}{2}\cdot\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\theta_{ij}^{(l)}\right)^2 \tag{13}$$

where L is the total number of layers in the neural network, s_l is the number of nodes in the l^{th} layer. Note that regularization is not done on the weights from the bias node. λ is the regularization parameter that helps us control the extent to which we want to penalize the neural network for the size of its weights.

Gradient descent is used to train the neural network, which means running several iterations making small adjustments to the parameters θ in the direction that minimizes the cost function. The weight update equation is shown in Eq. (14).

$$\theta^{(l)} := \theta^{(l)} - \alpha.\frac{\delta J}{\delta\theta^{(l)}}, l = 1, \ldots, L-1 \tag{14}$$

where α is the learning rate that controls the size of the adjustment, it must not be too small, else the learning will take very long, and it must not be too large, else the network will not converge. $\theta^{(l)}$ is a matrix and therefore the derivative term is also a matrix.

The *backpropagation* algorithm is used to calculate the derivative term. The intuition is to calculate the error term at every node in the network from the output layer backwards and use this to compute the derivative. We shall denote the error vector as $\boldsymbol{\delta}^{(l)}$ where l denotes the layer number. Eq. (15) denotes the error in the output layer of our example network.

$$\boldsymbol{\delta}^{(4)} = \boldsymbol{y} - \boldsymbol{a}^{(3)} \tag{15}$$

Except for the output layer the error term is defined as:

$$\boldsymbol{\delta}^{(l)} = \left(\theta^{(l)}\right)^T \boldsymbol{\delta}^{(l+1)} \diamond \sigma'\left(\boldsymbol{a}^{(3)}\right), \quad l \in \{2, \ldots, L-1\} \tag{16}$$

where $\sigma'(.)$ denotes the derivative of the sigmoid function, $\diamond$ signifies an element-wise multiplication and L is the total number of layers in the network, here $L = 4$. Note that $\boldsymbol{\delta}^{(1)}$ can be calculated but error on our inputs does not have any significance.

$$\sigma'\left(\boldsymbol{a}^{(3)}\right) = \sigma\left(\boldsymbol{a}^{(3)}\right) \diamond \left(\mathbf{1} - \sigma\left(\boldsymbol{a}^{(3)}\right)\right) \tag{17}$$

Eq. (18) is the derivative of the cost function (without the regularization term) computed using the errors previously found. The mathematical proof[2] of backpropagation is beyond the scope of this chapter. The final gradient averaged over all training examples with the regularization term is Eq. (19)

$$\frac{\delta J}{\delta \theta^{(l)}} = \boldsymbol{\delta}^{(l+1)} \left(\boldsymbol{a}^{(l)}\right)^T \tag{18}$$

$$\frac{\delta J}{\delta \theta^{(l)}} = \frac{1}{m} \cdot \sum_{i=1}^{m} \left(\boldsymbol{\delta}^{(l+1)} \left(\boldsymbol{a}^{(l)}\right)^T\right) + \lambda \cdot \theta^{(l)} \tag{19}$$

The weights of the network are *randomly initialized* to small positive or negative real values. If one were to initialize all the weights to the same value (say zero) then the gradient calculated at every node in a layer would be the same, and we'd end up with a neural network with lots of redundancies. Note that if there were no hidden layers this would not be the case but the power of the algorithm significantly goes down without them.

5.4. Limitations

Neural networks have large time and space requirements. Assume an n hidden layer fully connected neural network with m neurons in each hidden layer. We have $(n-1) \times m \times m$ parameters just in the hidden layers. This number is for the basic MLP network and more sophisticated implementations (*deep learning)* will have even more parameters.

For example, ILSVRC[3] evaluates algorithms for object detection and image classification at large scale to measure the progress of computer vision for large scale image indexing for retrieval and annotation. Over the years, deep neural networks have been used to solve the problem statement with better accuracy every year. AlexNet [5] had 60 million parameters and took two weeks to train on 2 GPUs in 2012 with 16.4% classification error using *convolutional neural networks*. GoogLeNet [6] had 4 million parameters achieving classification error of 6.66% in 2014 with the advent of their inception module in the *convolutional neural network*. So, even as the situation continues to improve, neural networks are still time and memory intensive.

[2]The concise proof can be found in chapter 2 of the book by Michael A. Nielson, "Neural Networks and Deep Learning", Determination Press, 2015

[3]http://www.image-net.org/challenges/LSVRC/

The second problem is predictability. Just as the human brain is an enigma that man has been trying to understand and find patterns in, the fact remains, humans are still unpredictable to quite an extent. Similarly, as far as machine learning algorithms go, neural networks are a black box and no one fully understands the functions that have been mapped in them once trained. Therefore no one can predict when they might fail and it is not always possible to justify the results produced as opposed to a rule-based classification method such as *decision trees.* Yet, neural networks have proved to be a great tool and are widely used by organizations today.

6. System design

Figure 5 depicts the block diagram of our system. The feedback is implied through the conglomerate controller/plant emulator artificial neural network (ANN) using the backpropagation algorithm.

The plant (quadrotor) block is simulated based on the model described in Section 2.

Two errors are generated:

$$\text{Model error: } y_{plant} - y_{model} \tag{20}$$

$$\text{Reference error: } y_{reference} - y_{plant} \approx y_{reference} - y_{model} \tag{21}$$

When approaching convergence, both errors tend to zero and the approximation in the reference error becomes increasingly accurate. The extended ANN has two functions – making an adaptive estimate of the next state of the plant given the current state and computing the control signals required by the plant to minimize reference error.

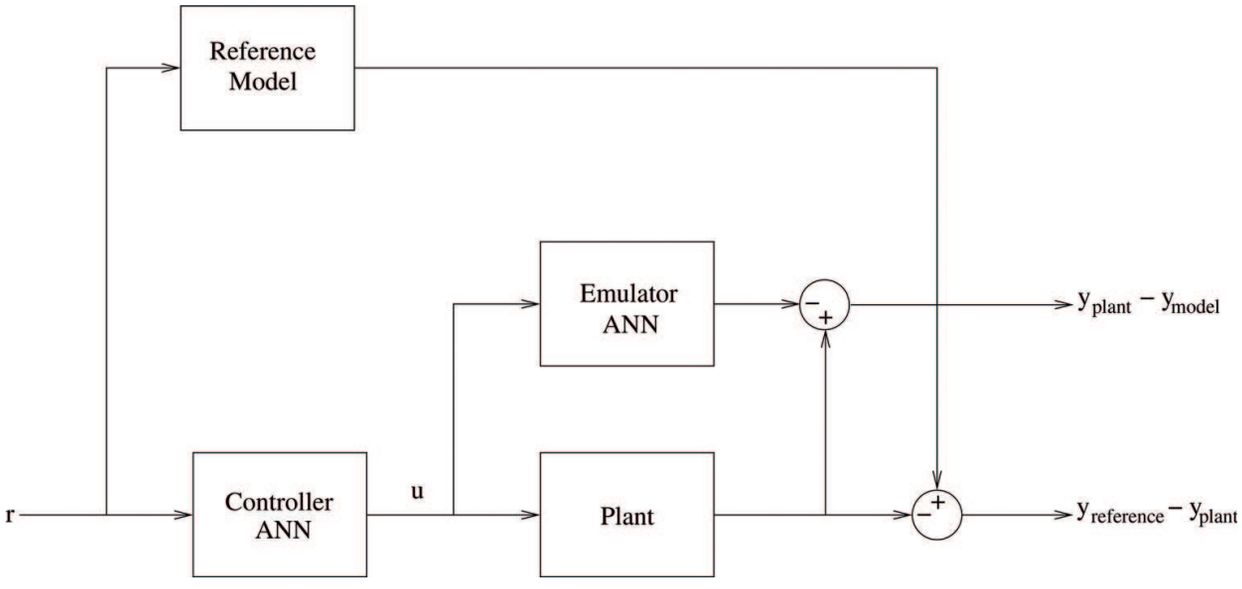

Figure 5. Block diagram of model reference adaptive control using artificial neural networks.

6.1. Selecting the reference model

The reference model consists of transfer functions for roll, pitch, yaw and z of the quadrotor. The transfer functions are as follows:

$$\text{roll and pitch: } G(s) = \frac{0.5}{s + 0.5} \tag{22}$$

$$\text{yaw: } G(s) = \frac{0.5}{s + 0.5} \tag{23}$$

$$\text{z: } G(s) = \frac{1}{s + 1} \tag{24}$$

The transfer functions are chosen to have a quick response to changing set points. Quadrotors are non-linear systems, yet we model the reference using first order transfer functions because we hold machine learning algorithms to the standard of an expert human operator. An expert helicopter pilot would not oscillate to attain a target *altitude* or *attitude* and neither should our controller.

6.2. Designing the plant emulator block

The plant emulator ANN predicts the next state of the plant given the control signal of the controller and the current state of the plant, thereby providing a channel for backpropagation to compute the errors on the control signals.

The plant emulator ANN needs to be pre-trained to a certain extent to ensure stability of the quadrotor while the controller ANN is learning in the air. Additionally the design of the plant emulator should be optimal for the application – accurate enough to model complexities and agile enough to respond quickly. Refer to **Figure 6** for the final plant emulator ANN.

To verify a good design for the plant emulator ANN, data was collected over several simulations run with varying set points for roll, pitch, yaw and z, separately and simultaneously. The control signals and plant states were mapped gathering a dataset of 8000 entries. This data was used to train the plant emulator ANN; hence set points were not mapped. In these simulations, de-centralized PID controllers were used for roll, pitch and yaw channels while a cascaded PID controller was used for the z channel.

The standard procedure in an offline setting is to divide the available tagged data into three parts (randomizing them if each entry is independent, which is not the case here) - the training set (~60%), the cross validation set (~20%) and the test set (~20%). The error on the cross validation set is monitored to select the hyperparameters (like λ or α) of the network and the test error (generalization error) is used to gauge its performance on unseen data. For the purpose of offline pre-training in this chapter, the true test is the actual system; hence we divide the data into training and cross validation sets only to make those design choices that are fixed in-flight.

6.2.1. Selecting the inputs and outputs of the network

The naive approach to this problem would be to demand all states of the plant – x, y, z, ϕ, θ *and* ψ, their derivates and double derivatives as output and to give as input, the previous

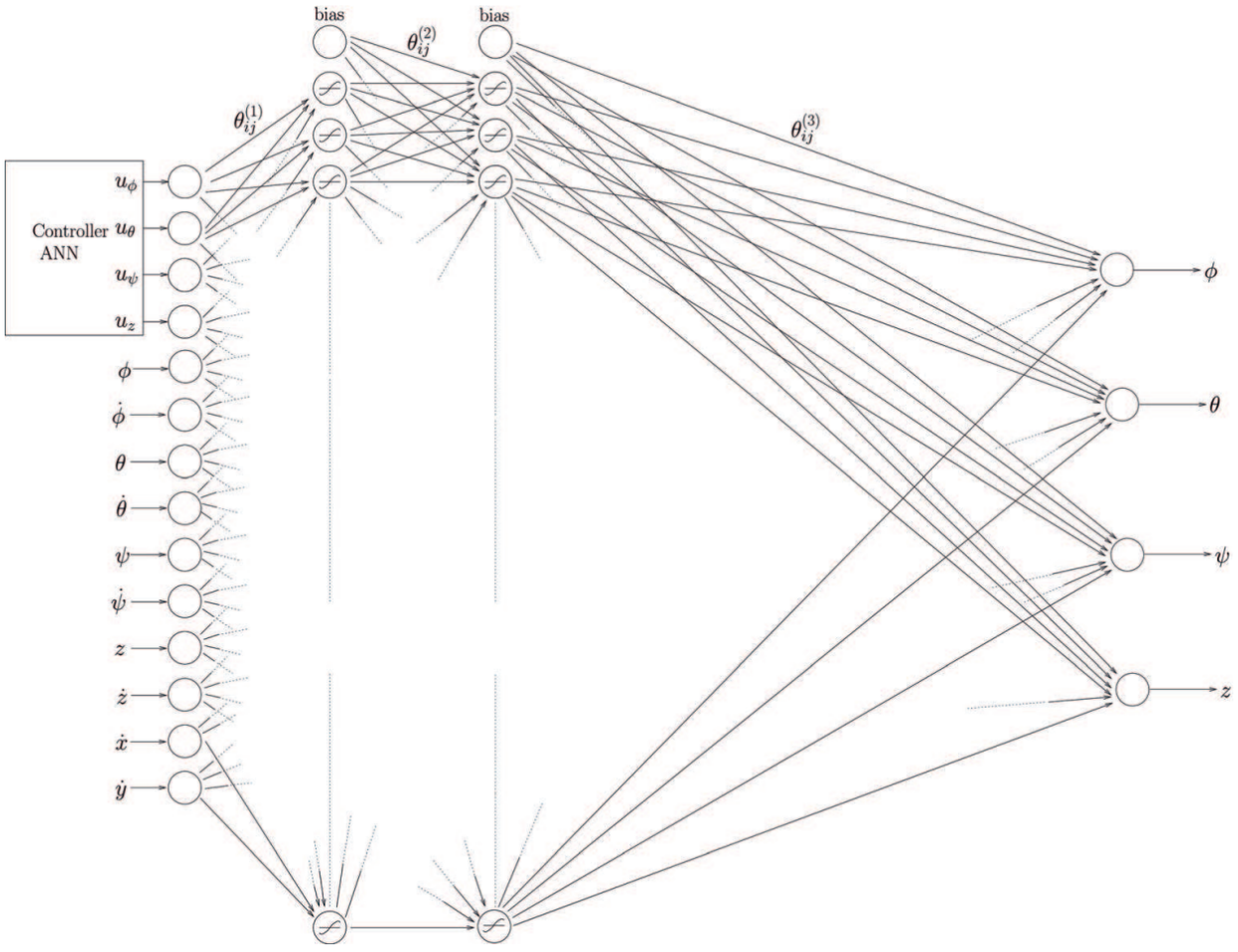

Figure 6. The double hidden layer plant emulator artificial neural network: First layer is the input, second and third layers are for computation, they comprise nodes with sigmoid activation applied and fourth layer is the output.

states of these 18 variables and the control signals. This playing-it-safe approach is costly as we place an undue burden on our algorithm.

Firstly – what is the output required from our network? The plant emulator should accurately predict the next state of the control variables of the system. Therefore the output should be the four control variables - z, ϕ, θ *and* ψ.

Secondly – what information would a human expert require to calculate the output? The previous states of z, ϕ, θ *and* ψ, their derivatives and the control signals would be required. Note that we do not need to give system information like the mass/inertia as input. The neural network will deduce such information based on the data. Interestingly, these 12 inputs are insufficient, $\dot{x}$ and $\dot{y}$ are required to model the dynamic thrust and drag coefficients by the model and therefore they must be given as inputs here.

If the initial choice of input/output variables is suboptimal leading to poor performance of the network, this step must be revisited.

6.2.2. Selecting depth of the network – accuracy vs. agility

A two hidden layer network was selected due to its ability to model most non-linear practical systems [7]. Increasing depth increases complexity of the network, which increases number of parameters and reduces agility of the network.

The field of *deep learning* is dedicated to using neural networks with deeper architectures, which are very powerful as we saw in Section 5.4. However, in deeper networks some of the design principles change, for example, in Section 5.3 the possibility of attenuation of information was pointed out in sigmoid-based neural networks, therefore in deeper implementations the *rectified linear unit* (ReLU) activation function is preferred. Architectural changes are also prevalent in deep learning such as *convolutional/recursive/recurrent* neural networks. Refer to Ref. [8] for more detailed reading on deep learning.

6.2.3. Selecting the width of the network

Performance with rectangular configurations in neural networks has been found to be equal to or better than performance with pyramid or inverted pyramid configurations [7] and therefore we have same number of hidden units in both hidden layers of our network.

As part of pre-training, the neural network performance was mapped against number of hidden units as seen in **Figure 7** and the elbow of the curve was found at 44 nodes (without

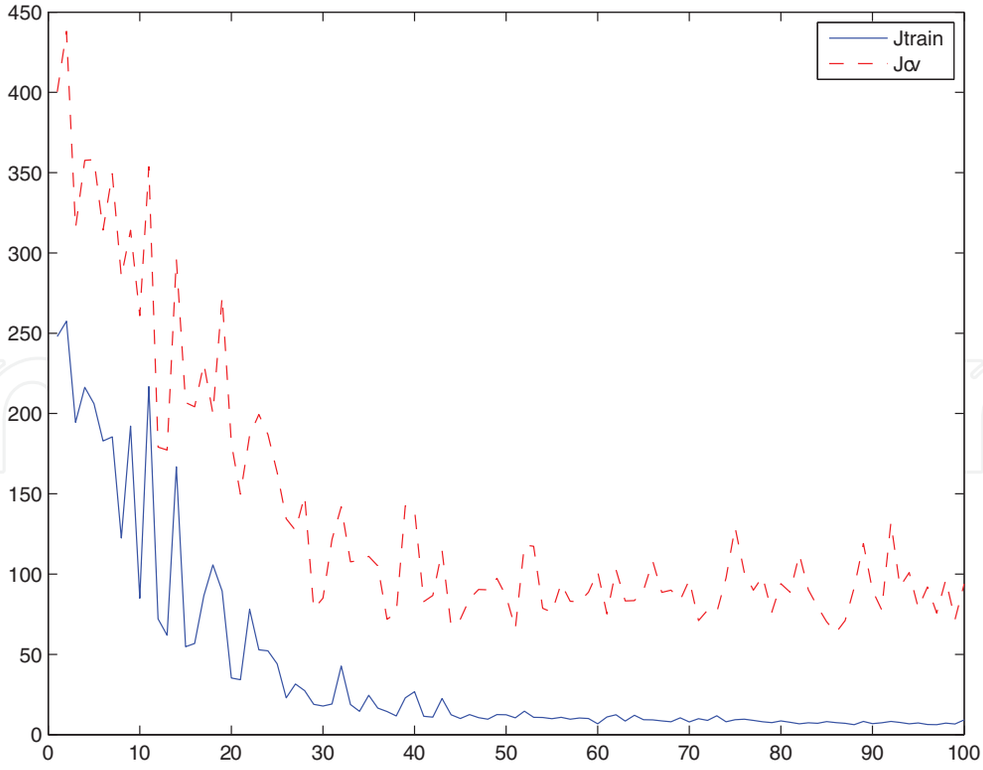

Figure 7. Costs on training and cross validation sets against hidden layer size.

the bias node) in each hidden layer. The elbow is the point beyond which the performance is unlikely to improve however the speed is sure to slow down with every node increased.

For this pre-training we stipulate to the cost function in Eq. (13) as we are training offline.

6.2.4. Cost function for online learning

To allow for real valued outputs, the ANN output layer has linear activation (read: no activation) applied while the hidden layers have sigmoid activation applied. The squared error cost function was used with regularization as shown in Eq. (25)

$$J = \frac{\left(y_{plant} - y_{model}\right)^2}{2} + \frac{\lambda}{2} \cdot \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(\theta_{ij}^{(l)}\right)^2 \tag{25}$$

Notice the difference with Eq. (13), there is no term in m as the learning happens with one data point at a time, i.e. online. The $\theta_{ij}^{(l)}$ term, which is a subset of the set of weights, must not be confused with θ the pitch angle of the quadrotor.

6.2.5. Backpropagation and learning

The backpropagation equations are as follows:

$$\text{error in final layer: } \boldsymbol{\delta}^{(L)} = \boldsymbol{y}_{model} - \boldsymbol{y}_{plant} \tag{26}$$

$$\text{error in hidden layers: } \boldsymbol{\delta}^{(l)} = \left(\theta^{(l)}\right)^T \boldsymbol{\delta}^{(l+1)} \diamond \boldsymbol{a}^{(l)} \diamond \left(1 - \boldsymbol{a}^{(l)}\right), l \neq L \tag{27}$$

$$\text{derivative: } \frac{\delta J}{\delta \theta^{(l)}} = \boldsymbol{\delta}^{(l+1)} \left(\boldsymbol{a}^{(l)}\right)^T + \lambda \cdot \theta^{(l)} \tag{28}$$

where $\diamond$ signifies element-wise multiplication. Notice the derivative term is Eq. (19) with $m = 1$. This distinction differentiates *stochastic* gradient descent from *batch* gradient descent (when we have all our data beforehand.) The parameter update equation is the same as Eq. (14).

The weights learned in offline pre-training were used to initialize the weights when actual testing was done. While the error was large to start with, the weights were biased in the correct direction. This is essential as the plant emulator ANN is the conduit for backpropagation to generate the error in the control signals and random corrections to the control signal based on a purely untrained plant emulator ANN right after take-off will destabilize and ground the craft before it has a chance to complete learning.

6.3. Designing the controller block

Figure 8 depicts the controller ANN. This segmented neural network does not resemble standard MLP networks as it is highly modified. This section will be structured differently from the previous one as it deals mainly with practical aspects and online learning.

We have a much broader understanding of a subject than we may be able to comfortably express in math. The easy approach to ML is to expect the neural network to learn

http://dx.doi.org/10.5772/intechopen.71487

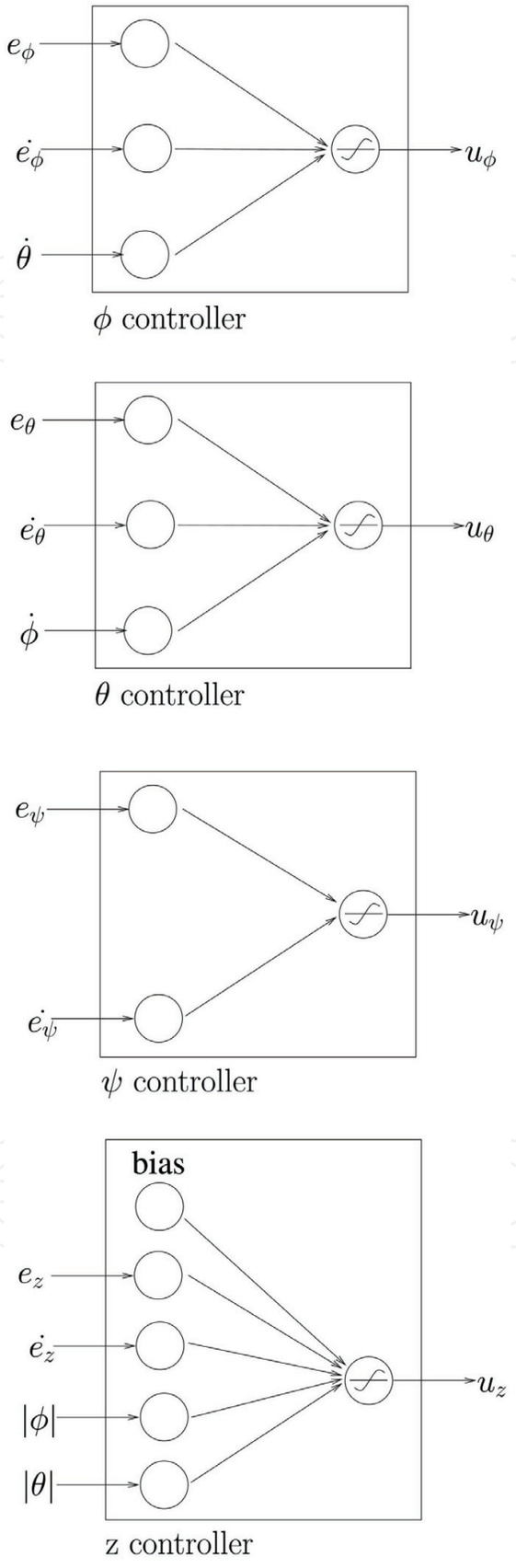

Figure 8. Segmented controller ANN with no hidden layers and constant multiplied constant offset sigmoid activation on output layer.

everything and we only provide the framework. However, this approach is unlikely to get the best results. For example, we know that a change in the z set point should not cause a change in the roll control signal, we can either make a fully connected neural network with hidden layers and expect the neural network to *learn* this by reducing the cross weights to zero or we can simply eliminate those connections ourselves. Taking the former approach revealed the learning was too slow and the quadrotor would take-off, somersault and crash as the changing z set point would cause the roll and pitch control signals to change too.

Such intuitions are application specific. It would greatly benefit the algorithm by reconciling our human understanding with the control system being designed.

The initial design of the controller ANN was very similar to the plant emulator ANN, it had two hidden layers, the inputs were the state of the plant and the state of the reference model, the outputs were the control signals which were unbounded and real valued. To summarize: 8 inputs, 4 outputs and 2 hidden layers.

At the very outset, it is clear that we should input the reference error directly rather than expecting the ANN to waste precious time learning that it has to minimize the difference between the state of the plant and the state of the reference model. Additionally an optimal number of nodes in the hidden layers cannot be experimented with as the testing is directly on the system (read: simulation) and the quadrotor would crash almost immediately after take-off. The adaptability of the controller is insufficient to keep up with the dynamically unstable nature of the quadrotor and the unbounded control signal gives noisy and, often, spurious commands to the motors thereby destabilizing the system.

A systematic approach was used to solve all the above issues:

1. The errors in the control variables were directly fed to the ANN, the second order effects were represented by appropriately adding inputs as per Eqs. (1–6).
2. The hidden layers were removed to make the ANN responsive and learn fast.
3. The output nodes were fed only those inputs that affected the state variable they controlled. Thus independence was achieved.
4. To make the control signal bounded, a constant-multiplied, constant-offset sigmoid function was used in the output layer.
5. A training regime was formulated so that each segment of the controller learned sequentially instead of simultaneously allowing for stability during online learning.
6. Feature scaling, gradient scaling and directed initialization were implemented specifically to aid in learning in the online setting.
7. The derivative term of the reference error was added to the cost function.

6.3.1. Controller summary

The controller ANN comprises four segments. The output node activations are:

http://dx.doi.org/10.5772/intechopen.71487

$$y_n = A.\frac{1}{1+e^{-(\theta_n)^T X_n}} + B = A.\sigma\left((\theta_n)^T X_n\right) + B \tag{29}$$

where A and B are real constants and $n \in \{1,2,3,4\}$ signifies each de-centralized controller.

1. **Z Controller**: This has a bias unit and four inputs - reference error in z, derivative of reference error in z, roll and pitch. Roll and pitch changes during a stable hover cause the quadrotor to deviate, to avoid this the absolute value of the corresponding angles are fed into the z controller. Its output is the z control signal. $A = 8$, $B = 4$.
2. **Roll Controller**: This has three inputs - reference error in roll, derivative of reference error in roll and derivative of pitch. The derivative of the pitch is to counter the coupling effect of roll and pitch. Its output is the roll control signal. A $= 0.01$, B $= -0.005$.
3. **Pitch Controller**: This has three inputs - reference error in pitch, derivative of reference error in pitch and derivative of roll. The derivative of the roll is to counter the coupling effect of roll and pitch. Its output is the pitch control signal. A $= 0.01$, B $= -0.005$.
4. **Yaw Controller**: This has two inputs - reference error in yaw and derivative of reference error in yaw. Yaw variation due to roll and pitch changes are minute enough to be compensated for by the controller. Its output is the yaw control signal. A $= 0.6$, B $= -0.3$.

6.3.2. Cost function and backpropagation

$$J = \frac{\left(y_{reference} - y_{plant}\right)^2}{2.{\sigma_r}^2} + \frac{\left(\dot{y}_{reference} - \dot{y}_{plant}\right)^2}{2.{\sigma_d}^2} + \frac{\lambda}{2}.\sum\nolimits_c^{\varphi\theta\psi z}\sum\nolimits_{i=1}^{s_l}\sum\nolimits_{j=1}^{s_{l+1}}\left(\theta_{ij}^{(c)}\right)^2 \tag{30}$$

σ_x denotes the standard deviation of the respective component and should not be confused with the sigmoid activation function. With this cost function, the rules have been bent. The derivative terms are not outputs of the neural network, yet we are attempting to minimize them. This modification proved to be a difference-maker in achieving online stability.

$$\text{error in final layer: } \boldsymbol{\delta}^{(L)} = \frac{\boldsymbol{y}_{plant} - \boldsymbol{y}_{ref}}{{\sigma_r}^2} + \frac{\dot{\boldsymbol{y}}_{\boldsymbol{plant}} - \dot{\boldsymbol{y}}_{\boldsymbol{ref}}}{{\sigma_d}^2} \tag{31}$$

$$\text{error in hidden layers: } \boldsymbol{\delta}^{(l)} = \left(\theta^{(l)}\right)^T \boldsymbol{\delta}^{(l+1)} \diamond \boldsymbol{a}^{(l)} \diamond \left(\mathbf{1} - \boldsymbol{a}^{(l)}\right), l \neq L \tag{32}$$

$$\text{error in control signals: } \boldsymbol{\delta}^{(c)} = A \cdot \left(\theta^{(c)}\right)^T \boldsymbol{\delta}^{(1)} \diamond \boldsymbol{a}^{(c)} \diamond \left(\mathbf{1} - \boldsymbol{a}^{(c)}\right) \tag{33}$$

$$\text{derivative: } \frac{\delta J}{\delta \theta^{(c)}} = \boldsymbol{\delta}^{(c)}\left(\boldsymbol{X}^{(c)}\right)^T + \lambda \cdot \theta^{(c)}, \quad c \in \{\varphi, \theta, \psi, z\} \tag{34}$$

λ is optional depending on the λ value used for the plant emulator ANN, i.e. if λ_{plant} is high (0.1) then $\lambda_{controller}$ can be set to zero as early stopping (the network stops learning as gradient becomes negligible) occurs within 6000 iterations. Of the 14 inputs to the plant emulator ANN,

only error in control signals are calculated. Here we see that the backpropagation calculation remains the same with two approximations:

1. $\boldsymbol{y}_{model}$ is the output of the network, not y_{plant} yet we backpropagate using y_{plant}
2. $\dot{y}_{plant}$ is being backpropagated through the network meant for y_{plant}

The reasons for the first approximation are twofold: firstly, we want the controller ANN to learn as if the plant emulator ANN got its prediction correct for better learning, and secondly, as the system converges $\boldsymbol{y}_{model}$ will very nearly equal $\boldsymbol{y}_{plant}$ making the approximation asymptotically accurate.

The reason for the second approximation is that it was experimentally found that without information of velocities the controller was not able to distinguish whether the plant was converging to or diverging from the reference when error was the same. This information was encoded into the controller by this modification and supplying the derivative of the reference error to each controller segment. Overhauling the network to incorporate the derivative terms as part of the output of the plant emulator ANN would make the system slower due to the increased complexity as it would have to learn the relation between a state and its derivative. This way the two error terms are additive and embody our intuition well - let us assume the error in z is moderate but the error in velocity of z is large, a larger correction would be effected in that iteration of learning, speeding it along. Conversely, if the reference error is large and the craft is approaching the reference too fast, the additive nature of the derivative term would negate any further positive adjustment due to the error in z thereby covering up the gap in intelligence.

6.4. Speed-ups for online learning

1. **Feature Scaling**: The inputs to the controller are divided by their respective standard deviations, since the values expected vary in magnitude and the weights of the network are regularized regardless of magnitude of input. In order to calculate the standard deviation in an online setting without losing significant information to truncation in data storage the following equations are used (as presented in Ref. [9]):

$$\text{Initialize } M_1 = x_1 \text{ and } S_1 = 0 \tag{35}$$

$$\text{Compute } M_k = M_{k-1} + \frac{(x_k - M_{k-1})}{k}, \quad S_k = S_{k-1} + (x_k - M_{k-1}) \cdot (x_k - M_k) \tag{36}$$

$$\text{Current value } \sigma = \frac{S_k}{k-1}, k \,\epsilon\, [2, n] \tag{37}$$

   The scaling is done selectively on the reference error and derivative of the reference error in each controller segment. This is done to scale up and give equal weight to the two errors while not magnifying higher order effects in the control signal.

2. **Directed Initialization**: Since there are no hidden layers and a single output for each segment there is no risk of redundancy in uniform initialization. Initializing all weights to

zero does not affect speed of convergence in comparison to random initialization in case of z and yaw. Therefore either can be used. For roll and pitch, learning is prone to instability and therefore we set the initial values of the weights to 1 or -1 depending on the sign of the weight required for stability. We have termed this *directed initialization*. This simplification of establishing parity in weights is effective as the inputs have been scaled.

3. **Gradient Scaling**: With the above two modifications, chances that the nodes, post sigmoid activation, will be saturated are high. Therefore, the gradient is divided by its σ (standard deviation) (calculated as shown in Point 1.) Gradient scaling is not compulsory for z.
4. **Sequence of Training**: The z and yaw controller weights are zero or random initialized. The pitch controller weights are directed initialized. The controllers are then trained sequentially. The order followed is:
   - **a.** The z set point is varied from zero to a non-zero hover value.
   - **b.** The pitch set point is varied from $-\frac{\pi}{3}$ to $\frac{\pi}{3}$.
   - **c.** The roll controller weights are set to the same values as the pitch controller.
   - **d.** The yaw set point is varied from $-\pi$ to π.

7. Performance evaluation

The testing was carried out as outlined above and the results were plotted. **Figure 9** depicts the first episode of training of the z controller, which converges within 10 seconds (at 100 Hz) and accurately follows the reference thereafter. $\alpha = 0.01$.

Figure 10 depicts the first episode of training of the pitch controller. Deviation from the reference is only due to the limitation on the maximum thrust the controller can demand (5 mN) due to bounding sigmoid function on control signal. $\alpha = 10$.

Figure 11 depicts the first episode of training of the yaw controller. Oscillations steadily decrease as training proceeds and the entire range of $-\pi$ to π is covered. $\alpha = 3$.

Figure 12 depicts simultaneous control of all four states, with controller weights continued forward from the first episode of training and plant ANN weights reset to initial estimate derived from offline training (accounting for the initial overshoot in z.) This result shows that the controller tracks well under simultaneous control signals from all four channels.

Figure 13 demonstrates the controller's robustness to mass and moment of inertia variations. At $t = 25$ sec the mass of all four motors are increased by 50 g leading to a 40% increase in total mass. The plant emulator learns these changes mid-flight enabling the controller to maintain tracking. The deviations in yaw are due to the inability of motors to meet increased thrust demands with increased moment of inertia. At $t = 75$ sec the mass and moments of inertia are reset to original values.

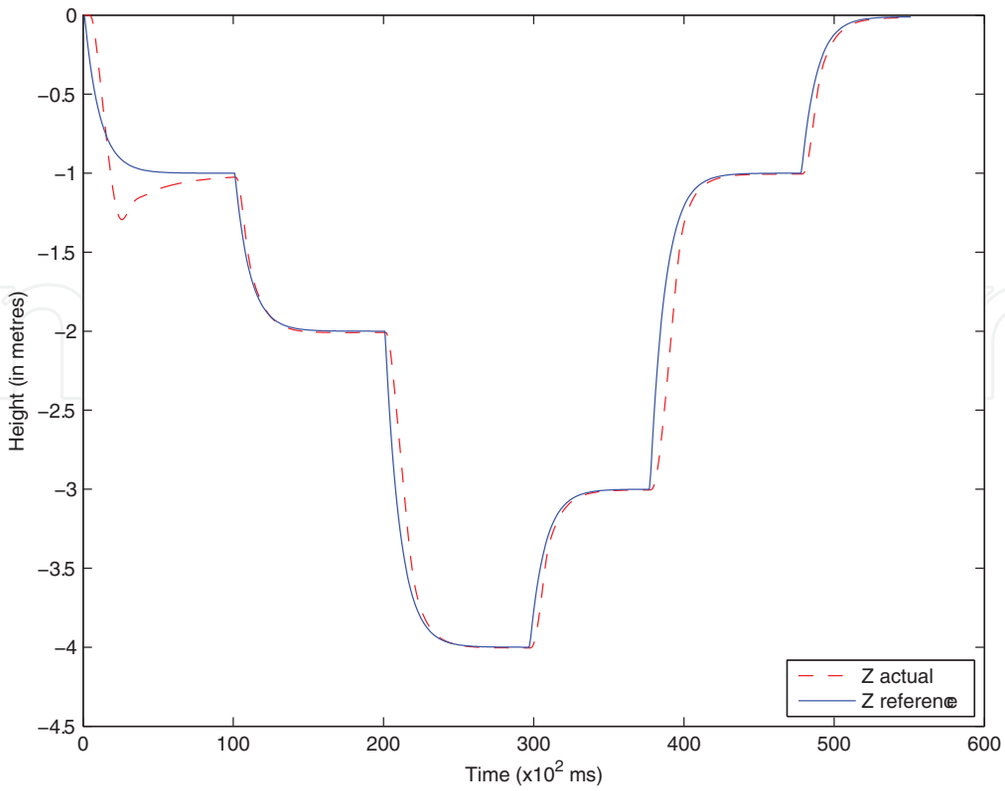

Figure 9. Starting with random weights the controller stabilizes the quadrotor to follow the reference.

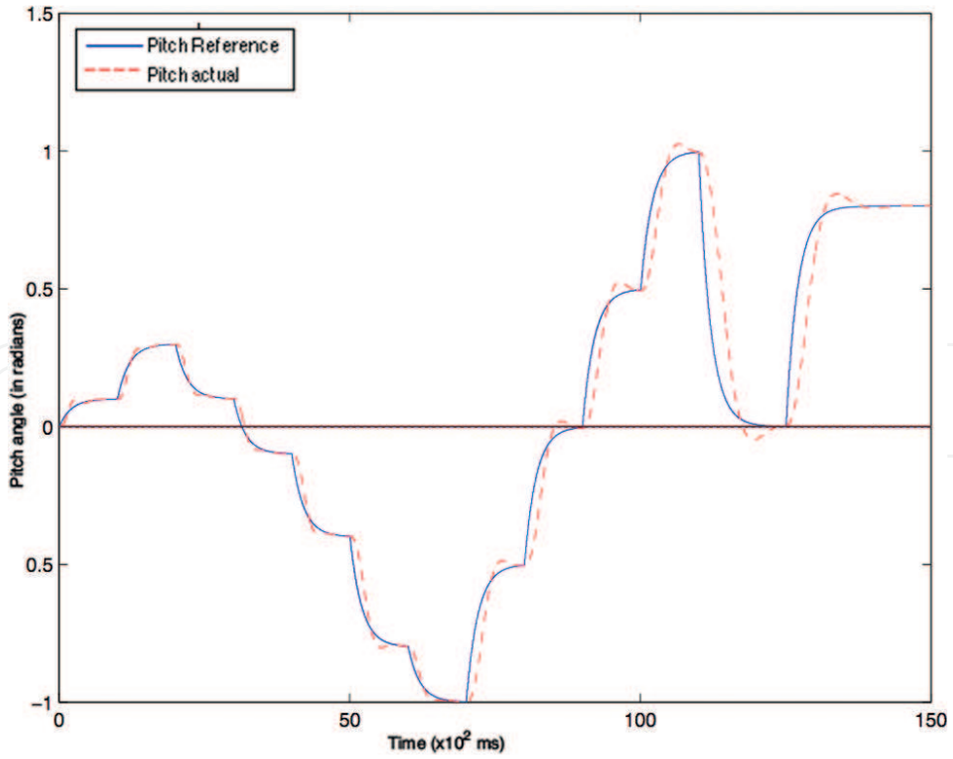

Figure 10. Directed initialized weights of pitch controller follows reference fairly well; any deviations occur due to thrust saturation.

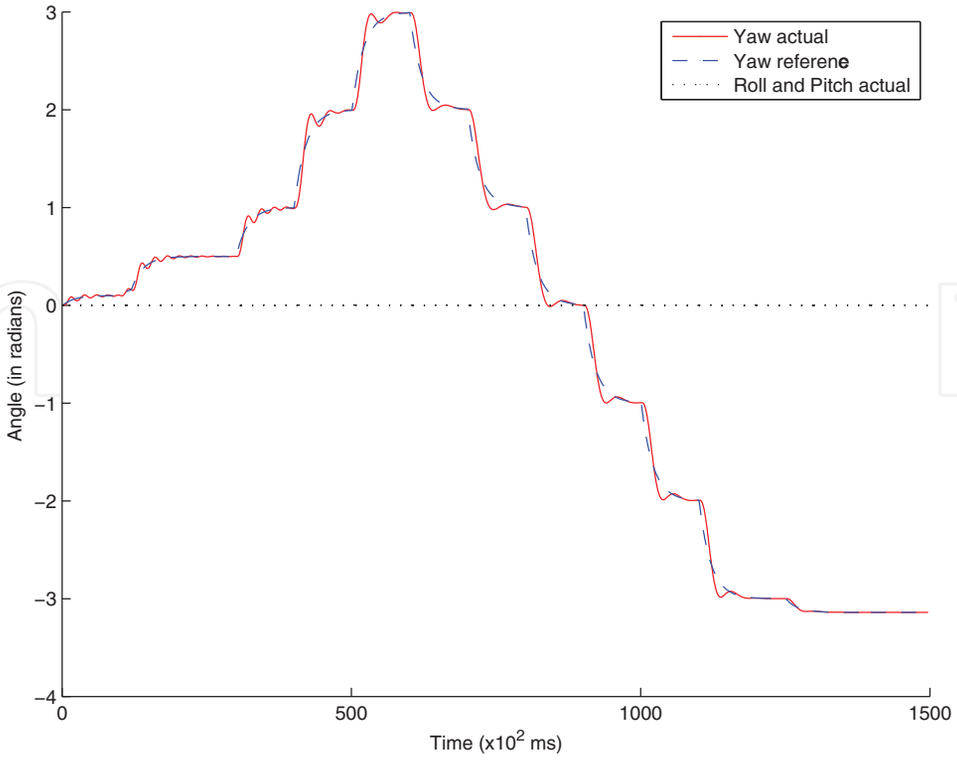

Figure 11. Zero initialized yaw controller reduces oscillations over time while tracking throughout.

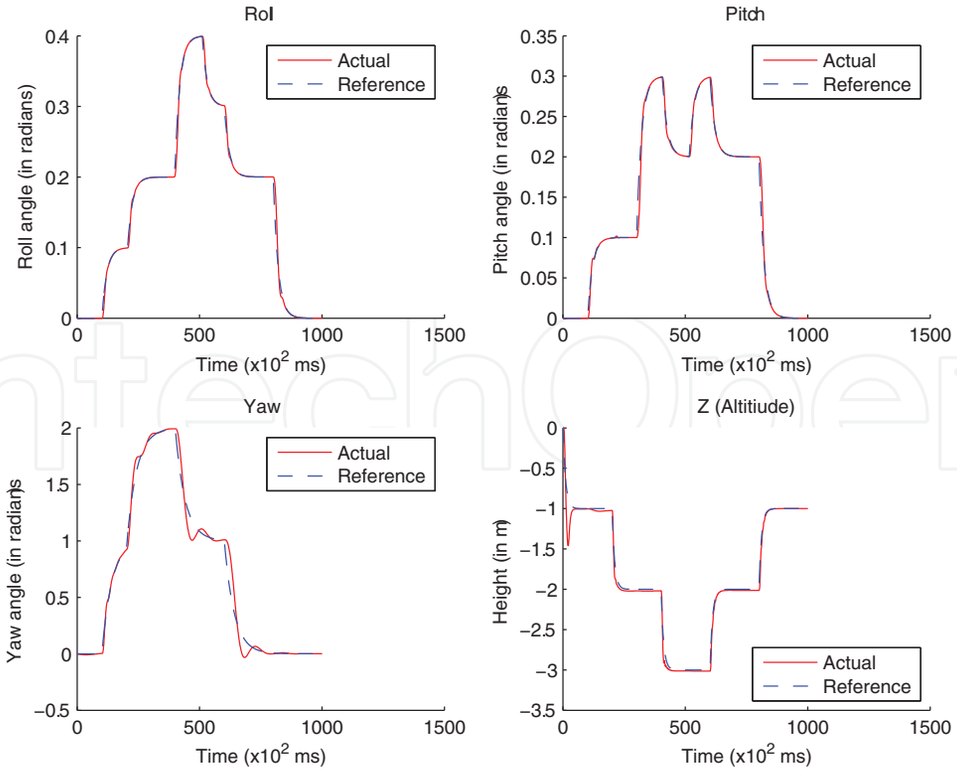

Figure 12. Simultaneous control of roll, pitch, yaw and z shows tracking with minimal error; yaw deviates slightly due to thrust saturation with large roll and pitch, however overshoot is acceptable.

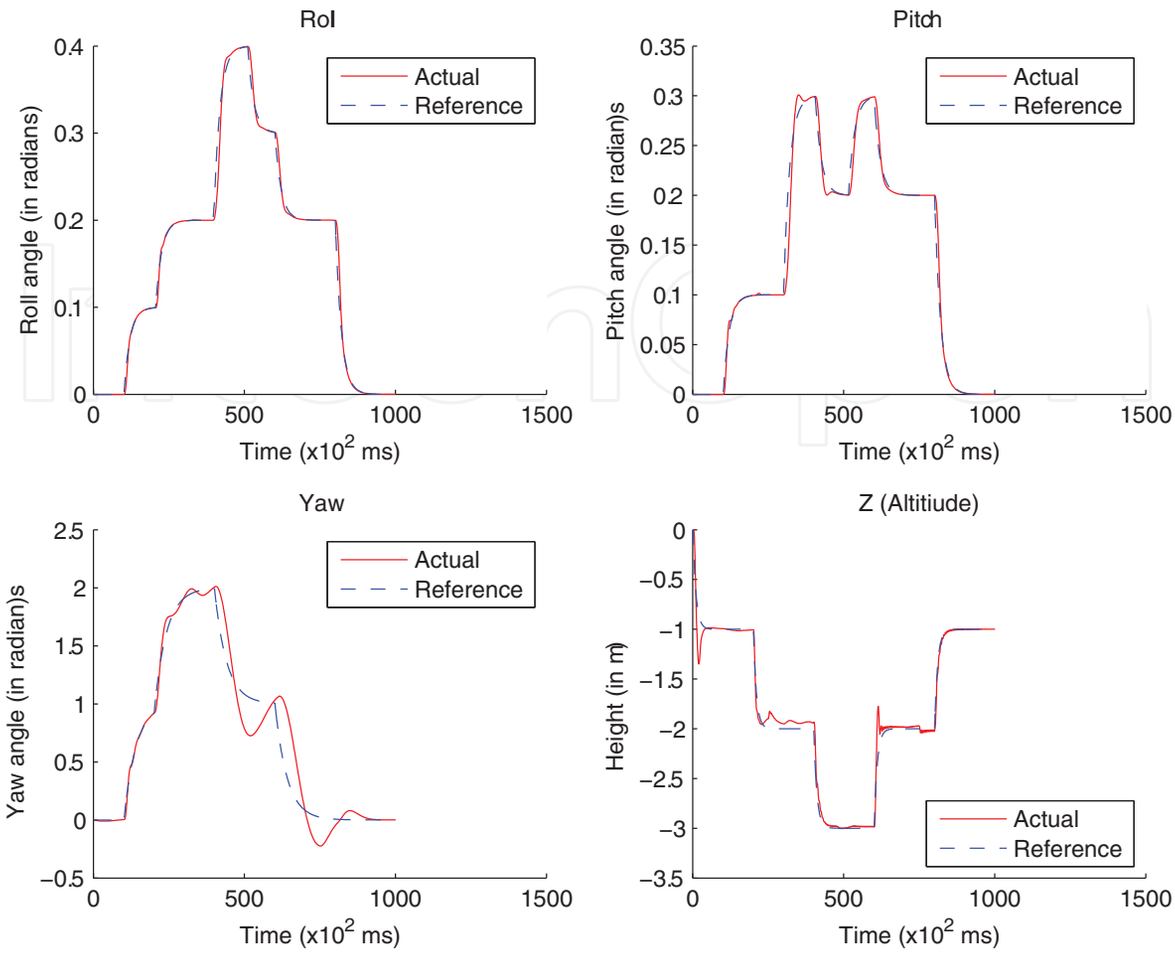

Figure 13. At $t = 25$ sec, the mass of all four motors is increased by 50 g; at $t = 75$ sec, mass and moments of inertia are reset to original values; results indicate the adaptive nature of the controller.

8. In closing

The tracking results of the quadrotor UAV system show the versatility of the control strategy. Once a framework is designed it can be applied to different systems with the same dynamics, i.e. can be applied to any quadrotor UAV, within certain limits, and will converge to stability, indicating *robustness*. It also withstands varying system parameters during operation due to changing environmental conditions or payloads, indicating *adaptability*.

A well-established and highly cited SIMULINK model was used for the simulations to prove the feasibility and good performance of this control strategy on a quadrotor UAV system. The system dynamics incorporated in this model include dynamically modeled thrust and drag coefficients, more reflective of a real-world scenario. Going forward, a disturbance rejection study can be done and the controller can be run on hardware to test it under real-world conditions.

Design choices for the neural network in terms of depth, width and choice of inputs were made based on real (read: simulated) data. The methodical process for this was outlined and applied to the quadrotor UAV system to justify the decisions made.

In real world systems, such as in UAVs where on-board processing is limited, processing time is a major factor and several methods for speed-ups were discussed which are computationally light. Even dynamically unstable systems such as UAVs could be stabilized using an untrained controller in-flight by devising a training regime.

Acknowledgements

Thanks to Mr. Raghu Ramachandran for his tireless efforts day in and day out, without whom the project upon which this chapter is based would not have been successful.

Thanks to all the people at Navstik Labs for the healthy work culture they nurture, setting the foundation for greatness. Thanks to Mr. Nitin Gupta (CEO and Founder of Navstik Labs) for giving me the opportunity to start something new and for his support along the way.

Thanks to Prof. Andrew Ng, whose ML lectures[4] have been referred to in introducing machine learning and neural networks.

Thanks to Dr. Bina Bakshi, my mother, whose love, care and support kept me going through the challenging times.

Author details

Nikhil Angad Bakshi[1,2]*

*Address all correspondence to: nabakshi@iitkgp.ac.in; nikhilangadbakshi@gmail.com

1 Indian Institute of Technology, Kharagpur, India

2 Navstik Labs, Pune, India

References

[1] Bakshi NA, Ramachandran R. Indirect model reference adaptive control of quadrotor UAVs using neural networks. In: Intelligent Systems and Control (ISCO), 2016 10th International Conference; 7–8 Jan. 2016; Coimbatore, India: IEEE; DOI: 10.1109/ISCO.2016.7727123

[2] Bouabdallah S, Siegwart R. Full control of a quadrotor. In: Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference; Oct. 2007; pp. 153-158. DOI: 10.1109/IROS.2007.4399042

[4] http://cs229.stanford.edu/materials.html, https://www.coursera.org/learn/machine-learning

[3] Fay G. Derivation of the Aerodynamic Forces for the Mesicopter Simulation. California, USA: Stanford University; 2001

[4] Bouabdallah S. Design and control of quadrotors with application to autonomous flying [Thesis]. EPFL; 2007. DOI: 10.5075/epfl-thesis-3727

[5] Krizhevsky A, Sutskever I, Hinton GE. ImageNet Classification with Deep Convolutional Neural Networks. Advances in Neural Information Processing Systems 25; 2012. pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[6] Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A. Going Deeper with Convolutions. Computer Vision and Pattern Recognition (CVPR); 2015. URL: http://arxiv.org/abs/1409.4842

[7] Bengio Y. Practical Recommendations for Gradient-Based Training of Deep Architectures. CoRR.2012;abs/1206.5533. DOI: arXiv:1206.5533

[8] Goodfellow I, Bengio Y, Courville A. Deep Learning. MIT Press; 2016. URL: http://www.deeplearningbook.org

[9] Welford BP. Note on a method for calculating corrected sums of squares and products. Technometrics. 1962:419-420. DOI: 10.2307/1266577