Modular Deep Recurrent Neural Network: Application to Quadrotors

Nima Mohajerin  
Dept. of Mechanical and Mechatronics Engineering  
University of Waterloo, Waterloo, ON, Canada  
nima.mohajerin@uwaterloo.ca

Steven L. Waslander  
Dept. of Mechanical and Mechatronics Engineering  
University of Waterloo, Waterloo, ON, Canada  
stevenw@uwaterloo.ca

Abstract—A modular deep Recurrent Neural Network (RNN) is introduced to facilitate the process of deploying various architectures of RNNs, and to automatically compute derivatives for gradient-based learning methods. The modularity leads to a set of new architectures, one of which includes feedforward inter-layer connections. By adding feedforward inter-layer connections in a multi-layer RNN, it is observed that the capability of the RNN to learn and model high-order dynamics and nonlinearities is significantly improved. The problem of vanishing/exploding gradient in space for a multilayer RNN is also alleviated using feedforward connections. These results are demonstrated using a quadrotor case study, for which a model of the altitude dynamics is learned with our particular network structure, while existing methods are unable to generalize as quickly or at all.

I. INTRODUCTION

Recurrent Neural Networks have been proposed for modelling dynamical systems using empirical data [1], [2]. Among various architectures, Nonlinear AutoRegressive with eXogenous input models (NARX) (and similar architectures, such as Nonlinear AutoRegressive Moving Average with eXogenous inputs - NARMAX) and Recurrent MultiLayer Perceptron (RMLP) have received a great amount of attention [3], [4], [5]. It has been shown that NARX is capable of learning dynamical systems [6]. In [3], the authors compare NARX with RMLP and a Memory Neuron Network (MNN) for identification of helicopter dynamics, and demonstrate that NARX is the most suitable method. However, the selection of number of delays poses a particular challenge in implementation [7], as a fairly good knowledge of the system to be modeled is required a priori in order to define sufficient delay units on a NARX input and output without over-specifying the network [3].

Other RNN architectures have also been explored for dynamic system modelling [8], [5], [9], [4]. Most of the architectures, however, are made of at most two hidden layers and feedback is used only in the intra-layer. Although adding more layers to the RNN architecture will increase the capacity of the network to represent higher order and more complex nonlinearities, as in deep neural networks [10], [11], usually using multiple layers is avoided because of the so-called problem of vanishing/exploding gradient [12]. In fact, one of the main reasons to prevent the application of gradient-descent optimization methods to train RNNs is that as the error gradient is propagated both back in time and towards former layers inside the network, it either explodes or vanishes, leading to an ill-conditioned Jacobian and numerical instability in the optimization. This problem exists both in the time and space domains of an RNN.

The motivation for our work stems from the need to identify accurate models for small unmanned aerial vehicles (UAVs), such as quadrotor helicopters. In recent years, quadrotors use has grown rapidly, both in research and industry, due to their ease of use, low maintenance requirements and agility of motion. Additionally, quadrotors possess interesting and complex dynamics which remain challenging to model accurately through all phases of flight. More accurate models are essential to enabling more precise flight control, which in turn will allow quadrotors to be used closer to objects in the environment and at higher speeds.

There have been quite a few successful attempts to capture the main dynamics of the quadrotor using physical properties of the system [13], [14]. However, modelling all aerodynamic effects, sensor and actuator uncertainties and all other nonlinearities using pure physical attributes of the vehicle is a very difficult task and requires wind tunnel testing to accurately assess aerodynamic coefficients. Therefore, a modelling approach based on empirical data collected from typical flight experiments may significantly reduce the effort required to define accurate models of quadrotor vehicles. Because quadrotors are dynamic systems, clearly this approach needs to be capable of capturing both dynamic characteristics and nonlinearities of the vehicle model.

In this paper, we aim at addressing the problem of vanishing/exploding gradient (in space) to add multiple layers to RNNs and make them capable of modelling high-order dynamical nonlinear systems such as a quadrotor. We will formulate a MOdular DEep Recurrent Neural Network (MODERNN) architecture which covers a variety of recurrent architectures and facilitates the derivative calculation process. Additionally, we will demonstrate that the presence of feedforward connections throughout a multilayer RNN contributes to decreasing the vanishing/exploding gradient problem in space, or equivalently the network output Jacobian will not be ill-conditioned even with multiple layers. Consequently, ordinary gradient descent training methods, such as the Levenberg-Marquardt algorithm (LM) are still applicable to train MODERNN. Through comparison with NARX and RMLP, we will assess the performance of MODERNN on modelling altitude of a quadrotor.

This work proceeds as follows. In the next section (Section II), our proposed RNN architecture is presented in detail. A modular method to obtain the derivatives of the network output(s) with respect to the network weights is presented afterwards in Section III. Then the learning algorithm for training MODERNN is described in Section IV. In Section V,
the simulation results of MODERNN modelling the altitude of a quadrotor are presented and compared with NARX and regular Multilayer RNN (MRNN). Finally, Section VI will conclude the paper and outline future work.

II. MODULAR DEEP RECURRENT NEURAL NETWORK

Our network definition is a generalization over RMLP with locally recurrent layers \( G_i \) and connections between all layers. An example of a three layer MODERNN is illustrated in Fig. 1. Each block \( G_i \) represents a locally recurrent layer. In fact, \( G_i \) is a dynamic MIMO system with \( m_i \) inputs and \( n_i \) outputs. The equations governing the dynamic of \( G_i \) are as follows:

\[
\begin{align*}
   x_i(k) &= A_i y_i(k-1) + B_i u_i(k) + b_i \\
   y_i(k) &= f_i(x_i(k)),
\end{align*}
\]

(1)

where \( k \) is the discrete-time index, and:

- \( x_i(k) \in \mathbb{R}^{n_i} \) is the state of the layer,
- \( y_i(k) \in \mathbb{R}^{n_i} \) is the output of the layer,
- \( u_i(k) \in \mathbb{R}^{m_i} \) is the input to the layer,
- \( A_i \in \mathbb{R}^{n_i \times n_i} \) is the feedback weight matrix,
- \( B_i \in \mathbb{R}^{n_i \times m_i} \) is the input weight matrix,
- \( b_i \in \mathbb{R}^{n_i} \) is a bias weight vector,
- \( f_i(\cdot) \) is the layer activation function,
- \( n_i \) is the number of the neurons inside the layer,
- \( m_i \) is the number of input signals to the layer.

We will refer to \( x_i(k) \) and \( y_i(k) \) as the network states. Also, we will refer to the vectorized form of the weights inside the layer \( G_i \) by \( p_i \):

\[
   p_i^T = \begin{bmatrix}
   B_i(:,1)^T & \cdots & B_i(:,m_i)^T \\
   A_i(:,1)^T & \cdots & A_i(:,m_i)^T \\
   \end{bmatrix} b_i^T,
\]

(2)

where \( A_i(:,j)^T \) is the transpose of the \( j \)-th column of \( A_i \) and similarly for \( B_i \). The vector \( p_i \in \mathbb{R}^{q_i} \), where \( q_i = n_i(m_i + n_i + 1) \) is the number of all weights inside \( G_i \). Note that \( u_i(k) \) includes the independent inputs (i.e. \( u(k) \)) as well as outputs from other layers (\( y_j(k-1), j \neq i \)). Note also that the output of a layer is actually the output of all neurons inside the layer.

Although we can include \( y_i(k-1) \) inside \( u_i(k) \) and make the equations more compact, we avoid this for two reasons. First, our proposed formulation is more modular, hence, easier to implement. Second, the process of calculating the derivatives is more straightforward, as will be seen shortly.

With inclusion of a connection matrix we can modify the inter-layer connection in a MODERNN. The proposed connection matrix, \( C \), is an \((L + 2) \times L \) matrix for an \( L \) layer network. Elements of \( C \), i.e. \( c_{ij} \), represent absence/presence of a connection between layer \( i \) and \( j \), by being either 0 or 1, where row \( L + 1 \) corresponds to connections from the independent input signals (\( u(k) \)) and row \( L + 2 \) corresponds to connections forming the network output signal. For example, the connection matrix for the example shown in Fig. 1, with fully connected sublayers, feedforward to each layer and network output dependent on only the last layer’s outputs, is given by,

\[
   C = \begin{bmatrix}
   1 & 1 & 1 \\
   0 & 1 & 1 \\
   1 & 0 & 0 \\
   0 & 0 & 1 \\
   1 & 0 & 0 \\
   0 & 0 & 1 \\
   \end{bmatrix}.
\]

(3)

Using this connection matrix, various architectures are attainable. For instance, to have an RMLP with 4 layers, the following connection matrix is used,

\[
   C = \begin{bmatrix}
   1 & 1 & 0 & 0 \\
   0 & 1 & 1 & 0 \\
   0 & 0 & 1 & 1 \\
   1 & 0 & 0 & 0 \\
   0 & 0 & 1 & 0 \\
   \end{bmatrix}.
\]

(4)

Note that in an RMLP the layers are locally recurrent and sequentially connected, i.e. each layer is only connected to itself and the next layer. Therefore, in equation (4), \( c_{i,i} = 1 \) for \( i = 1, 2, 3 \) correspond to local feedback and \( c_{i,i+1} = 1 \) for \( i = 1, 2, 3 \) correspond to the sequential connection. Also, \( c_{4,1} = 1 \) means that the independent input is only connected to the first layer and \( c_{6,1} = 1 \) means that the network output is the output from the final layer only.

Other forms of architectures are also attainable. For example, the architecture shown in Fig. 2 is a MODERNN in which the outputs are derived from the first and last layers. The connection matrix corresponding to this architecture is,

\[
   C = \begin{bmatrix}
   1 & 1 & 1 \\
   1 & 1 & 1 \\
   1 & 1 & 1 \\
   1 & 0 & 1 \\
   \end{bmatrix}.
\]

(5)

As derivatives are required for each network definition to employ gradient-based learning methods, we present a generalized output Jacobian calculation, for all architectures derived from MODERNN using a connection matrix similar to what we already presented. However, we will use MODERNN to refer to a network with all inter-layer connections present, because we will observe that the presence of inter-layer connections is a crucial fact to enhance the network training process. Such a network with \( L \) layers is presented as a function that maps inputs (\( u(k) \in \mathbb{R}^{m_i} \)) to outputs (\( y(k) \in \mathbb{R}^{n_i} \)):

\[
y(k) = \Omega_L(u(k)).
\]

(6)

For a more compact formulation, assume the input sequence is defined over a time horizon, \( T \), i.e., \( k = 1, \ldots, T \). Then,
which corresponds to the derivatives of the layer weights with \( \Gamma \) is the Jacobian of the inputs to that layer.

We can arrange \( u(k) \) (for all \( k = 1, \ldots, T \)) inside a matrix \( U \in \mathbb{R}^{m \times T} \) and pass it to the network.

The output becomes a matrix \( Y \in \mathbb{R}^{n \times T} \) where the \( k \)th column of \( Y \), i.e. \( y(k) \) is calculated by equation (6). 

III. NETWORK JACOBIANS FOR MODERNN

In this section, we present a modular method to derive the network Jacobians. It has been reported that for different architectures, obtaining network output derivatives is a time consuming process [15]. We attempt to address this problem and speed up the process of modifying/deploying present/new architectures.

To calculate the network Jacobians, we form \( p \in \mathbb{R}^q \) which is a vector encompassing all the weights inside a MODERNN network, i.e.,

\[
p^T = [p_1^T \cdots p_L^T], q = \sum_{i=1}^L q_i,
\]

where \( L \) is the number of locally recurrent layers inside the network whose indexes are sorted in ascending order from input towards output. Additionally, the number of all weights inside this network is given by \( q \).

Now, considering an arbitrary layer, \( G_i \), we can write the Jacobian update rule as follows:

\[
J^p_i(k) = \frac{\partial y_i(k)}{\partial p} = \text{diag}( f'(x_i(k))) \left( A_i J^p_i(k) + B_i J^y_i(k-1) + \Gamma_i(k) \right).
\]

In equation (9), \( J^p_i(k) \in \mathbb{R}^{n_i \times q} \) is the Jacobian of the outputs of the layer \( G_i \) at time \( k \) and \( J^y_i(k) \in \mathbb{R}^{n_i \times q} \) is the Jacobian of the inputs to that layer. \( \Gamma_i(k) \) is a matrix which corresponds to the derivatives of the layer weights with respect to the network weights,

\[
\Gamma_i(k) = \frac{\partial A_i}{\partial p} u_i(k) + \frac{\partial B_i}{\partial p} y_i(k) + \frac{\partial b_i}{\partial p}.
\]

Note that equation (9) is a recursive update for \( J^p_i(k) \). Therefore, we need to have an initial value for it at time \( k = 0 \). Usually, because training is started from a stationary point, we assume zero initial condition for this matrix. Then, we need to form two matrices: \( J^p_i(k) \) and \( \Gamma_i(k) \). To form \( J^p_i(k) \), remember that

\[
u_i(k) = \begin{bmatrix} u^T(k) & y^T_i(k-1) & \cdots & y^T_{i-1}(k-1) & y^T_{j+1}(k-1) & \cdots & y^T_L(k-1) \end{bmatrix}^T.
\]

Therefore, we see that

\[
m_i = m + \sum_{j=1}^L n_j,
\]

Because the independent inputs are not dependent on the network weights, the first \( m \) rows of \( J^p_i(k) \) are zero.

\[
J^p_i(k) = \begin{bmatrix} \mathbf{0}_{m \times q}^T & J^y_i^T(k-1) & \cdots & J^y_{i-1}^T(k-1) \\
A_i & \mathbf{0}_{q \times q} & \cdots & \mathbf{0}_{q \times q} \\
\mathbf{0}_{q \times q} & A_i & \mathbf{0}_{q \times q} & \cdots & \mathbf{0}_{q \times q} \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
\mathbf{0}_{q \times q} & \mathbf{0}_{q \times q} & \cdots & A_i & \mathbf{0}_{q \times q} \\
\end{bmatrix}.
\]

The remaining elements inside the Jacobian \( J^p_i(k) \) are all calculated at the previous time step, and so this term can be determined recursively.

To modularly formulate \( \Gamma_i(k) \), first let us define another matrix \( \Lambda_i(k) \) for each layer, \( G_i \), as follows:

\[
\Lambda_i(k) = \frac{\partial A_i}{\partial p} u_i(k) + \frac{\partial B_i}{\partial p} y_i(k) + \frac{\partial b_i}{\partial p}.
\]

In other words, \( \Lambda_i(k) \) plays the same role as \( \Gamma_i(k) \) but when the derivative is taken with respect to local weights only. Note, \( \Lambda_i(k) \) is in \( \mathbb{R}^{n_i \times q} \). With some algebraic manipulation, the term \( \Lambda_i(k) \) can be presented in the following form:

\[
\Lambda_i(k) = [\Lambda_{i,1}(k) \quad \Lambda_{i,2}(k) \quad \mathbf{I}_{n_i}]_{n_i \times q_i},
\]

where

\[
\Lambda_{i,1}(k) = [u_{i,1}(k) \mathbf{1}_{n_i} \quad u_{i,2}(k) \mathbf{1}_{n_i} \quad \cdots \quad u_{i,m_i}(k) \mathbf{1}_{n_i}],
\]

and \( u_{i,1}(k) \) refers to the first element inside \( u_i(k) \), and so on.

Now that we can calculate \( \Lambda_i(k) \) for all layers, it is possible to form \( \Gamma_i(k) \):

\[
\Gamma_i(k) = [\mathbf{0}_{n_i \times a} \quad \Lambda_i(k) \quad \mathbf{0}_{n_i \times b}].
\]

For equation (17), we define

\[
a = \sum_{j=1}^{i-1} q_j, \quad b = \sum_{j=i+1}^L q_j
\]

Now that \( \Gamma_i(k) \) is defined, we can return to equation (9) and recursively update the Jacobian.

IV. A LEARNING ALGORITHM FOR TRAINING MODERNN

To address the problem of training RNN using network Jacobians, there are two commonly used methods: Back-Propagation Through Time (BPTT) and Real-Time Recurrent Learning (RTRL) [16], [17], [18]. The time-series data which we aim to train the network with are trajectories in time. For our particular problem, i.e. modelling of a quadrotor’s altitude dynamics, each training sample is a 2-tuple of a motor input (sum of motor speeds) and a position output (altitude of the vehicle), both over time, i.e. \( (u_d(k), z_d(k)) \). Because higher accuracy of the trained model is our final goal, we adopt an off-line, batch learning method to expose the network to more information. One of the most commonly used approaches
Sum-of-Squared Errors (SSE) cost function: 

\[ E = 0.5 \sum_{k=1}^{T} (y(k) - z_d(k))^2 = 0.5e^T e, \]  

(19)

where: 

\[ e^T = [e(1) \quad e(2) \quad \ldots \quad e(T)]^T, \]  

\[ e(k) = y(k) - z_d(k), \quad \text{for } k = 1, \ldots, T, \]  

(20)

Note that in our setup, the network has one output, which is the approximated altitude. We can have more than one, for example in case of a full quadrotor modelling, which involves more effort in computing the Jacobians.

**Remark:** Because MODERNN has a recursive nature, it is necessary to initialize it properly. As the evolution of the network states is not known *a priori* and is highly dependent on the network weights, initializing the untrained network to start from a non-zero initial condition is not reasonable as the network weights are merely random values at that point in the optimization. However, we propose that at the first stages of the training process, for each training sample, the network is initialized to zero. The training sample, i.e. the altitude envelop, should also start from a stationary point (e.g. quadrotor landed on the floor). The network weights should then converge to a set of more representative values. The training samples at these stages should encompass enough information from the beginning of a flight (e.g. both during take-off and with some variable altitude flight). Then, as the network is trained on these primary flight data, we can proceed to expanding the training horizon. For this work, because we are more interested in the MODERNN architecture and its capabilities, we focus on training the network with zeroed initial conditions.

LM is essentially a second-order optimization method with variable step size which approximates the Hessian of the cost function [20]. At each training iteration \( k' \), the update rule in LM is given by,

\[ \Delta p(k') = -(J^T J + \lambda I)^{-1} J^T e. \]  

(21)

In equation (21), \( J \) is the Jacobian of the cost function over time given by,

\[ J = \begin{bmatrix} \frac{\partial e(1)}{\partial p_1} & \frac{\partial e(1)}{\partial p_q} \\
\vdots & \vdots \\
\frac{\partial e(T)}{\partial p_1} & \frac{\partial e(T)}{\partial p_q} \end{bmatrix}_{T \times q} = \left[ \begin{bmatrix} \frac{\partial e(1)}{\partial p} \\
\vdots \\
\frac{\partial e(T)}{\partial p} \end{bmatrix}^T \right] = \frac{\partial e}{\partial p} \]  

(22)

We propose a training method that divides the optimization into four nested loops. The outer most loop, \( l_1 \), handles the choice of training and validation samples. The two middle nested loops, \( l_2 \) and \( l_3 \), perform the LM optimization over the selected training set with \( n_v \)-fold cross-validation. The inner most loop, \( l_1 \), is to modify the parameter \( \lambda \) using equation (21).

Assume that the network is properly initialized and the dataset has \( n_D \) samples in it, each is an input/output trajectory segment with \( T \) time steps:

\[ D = \{(u_d, z_d)\}, |D| = n_D, \]  

(23)

where \(|\cdot|\) states the cardinality of a set and

\[ u_d = [u_d(1) \ldots u_d(T)], z_d = [z_d(1) \ldots z_d(T)]. \]  

(24)

At each iteration of \( l_1 \) we pick \( n_s = n_{tr} + n_v \) samples from \( D \) to form the subset \( D_S \).

\[ D_S \subset D, |D_S| = n_s \]  

(25)

The number of trajectory segments that we simultaneously train the network on is \( n_{tr} \) and \( n_v \) is the number of trajectory segments that we validate the network on. Both constants should be kept small to reduce the computational complexity of the training process. Then, at each iteration of \( l_2 \), we divide \( D_S \) into two sets: the training set, \( D_{tr} \), and the validation set, \( D_v \), so that \( |D_{tr}| = n_{tr} \) and \( |D_v| = n_v \).

Having the training and validation sets, the optimization starts in loop \( l_3 \) and continues until the validation fails, i.e. the error over validation set starts to increase. At each iteration of \( l_3 \), we run the network over the entire \( D_{tr} \), update and collect the network Jacobians, as described in Section III, and calculate the errors. Then we form the Jacobian in equation (21) as described in equation (22). The weight update process is performed in loop \( l_4 \), where we update the network weights using equation (21) starting with an initial (usually small) \( \lambda_0 \). At each iteration of \( l_4 \), we check if the error is decreased/increased and update \( \lambda \) accordingly. The detailed steps of this algorithm are defined in Algorithm 1. Although specific choices are made in terms of stopping criteria, step size, etc., these choices are not a requirement of the algorithm and may easily be modified for application of the MODERNN algorithm to other problem instances.

It is important to note that each weight update is performed using the Jacobian that is computed over a number of flights at once. We do not apply each training sample inside \( D_{tr} \) individually, but instead carry out the training simultaneously over the entire set, \( D_{tr} \). The function Minibatch is so named to refer to the training over the entire batch of trajectory segments, \( D_{tr} \). The formulation of MODERNN and the presented basic algorithm is quite flexible both in terms of attainable architectures and applicable training methods, as the network Jacobians can be computed explicitly. In the version of MODERNN with all inter-layer connections, the problem of vanishing/exploding gradient is less severe because direct connections from the output layer(s) to all other layers facilitate the error information transfer between layers. For function approximation problems, the only information that the network receives from outside the network is the Sum-of-Squared Errors. In traditional multilayer networks, the error information starts to flow from the output layer through the middle layers back to the input layer in a sequential manner. At each layer this information is either attenuated or amplified, and because network weights are usually initialized in the range \([-1, 1]\], attenuation occurs. In the MODERNN, the error information travels through a number of different paths, this multipath transfer of information contributes to faster and better learning and less attenuation of the error gradient in former layers, as presented in Section V.

**V. Simulation Results**

We apply the MODERNN structure to the problem of learning quadrotor altitude dynamics. The details of quadrotor
is a simple model for ground effect. The ground effect acts as input to equation (26) is not considered. Instead, each altitude trajectory was created by applying a random sum of 10 sinusoids with variable frequencies in range of $[1, 10] Hz$ as input. To capture the ground effect, altitude is varied in the range $[0, 2]$ meters over the whole dataset. The dataset is collected using a $f_s = 10 Hz$ sampling frequency. In Fig. 3 one sample of the dataset with normalized values is illustrated.

![Fig. 3: A generated data sample.](image)

Having generated the dataset, the MODERNN, RMLP and NARX architectures are trained on it. Our experiments on series-parallel learning of NARX using the MATLAB toolbox failed when we closed the loop for prediction. Instead, we implemented our own parallel-model NARX, calculated the analytic derivatives of the network and implemented a training process based on the same method we presented in Section III. We validated our implementation on examples given in [1]. It is therefore important to highlight that we do not use any form of teacher forcing in training the three networks; they are trained and evaluated in a closed loop fashion only.

The parameters under investigation are: the number of layers $L$, the number of hidden neurons in each layer $h$, and the size of the minibatch $n_{tr}$. For NARX, the number of delays over input and output $n_d$ is also investigated. Note that in multilayer cases, we choose to have the same number of neurons in all layers. Also, remember that the last layer has a linear output as is always the case for function approximation applications. The result are summarized in Table II. In this table, $E_m$ is the mean value of the errors over the entire dataset and $t_{tr}$ is the training time on an i7 Core machine. In general, we observe that finding a working configuration becomes increasingly harder from MODERNN to RMLP and to NARX. It was not possible to find a working architecture for NARX with $n_{tr} = 5, 10, 15$ and RMLP with $n_{tr} = 5, 10$, while MODERNN can learn on $n_{tr} = 5$ and higher. For each of the reported cases we run the training 5 times, each time with a different weight (random) initialization. The best results are reported only. As a result of the ability to

<table>
<thead>
<tr>
<th>$m$</th>
<th>$k_{ge}$</th>
<th>$h_{ge}$</th>
<th>$k_t$</th>
<th>$r_{max}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 kg</td>
<td>0.5</td>
<td>1 m</td>
<td>$1.95 \times 10^{-5}$ Ns$^2$/rad</td>
<td>2 m</td>
</tr>
</tbody>
</table>

**Algorithm 1: Learning Algorithm for MODERNN**

1. for $l_1 = 1$ to $\frac{n_{tr}}{n_{tr_{in}}}$ do $\triangleright$ loop $l_1$
   2. for $l_2 = 1$ to $\lceil S \rceil$ do $\triangleright$ loop $l_2$
      3. $D_{lr} \leftarrow D_S(S(l_2))$
      4. $D_v \leftarrow D_S^\prime(D_{lr})$
      5. $\gamma_1 \leftarrow 1$
      6. $\lambda \leftarrow \lambda_0$
      7. while $\gamma_1$ do $\triangleright$ loop $l_3$
         8. $e_{v,0} \leftarrow \text{MINIBATCH}(D_v, p, 0)$
         9. $[e_{tr,0}, J] \leftarrow \text{MINIBATCH}(D_{tr}, p, 1)$
      10. $\gamma_2 \leftarrow 1$
         11. while $\gamma_2$ do $\triangleright$ loop $l_4$
          12. $\Delta p \leftarrow -(J^T J + \lambda I)^{-1} J^T e$
          13. $e_{tr,1} \leftarrow \text{MINIBATCH}(D_{tr}, p + \Delta p, 0)$
          14. if $e_{tr,1} e_{tr,1} < e_{tr,0} e_{tr,0}$ then
             15. $\lambda \leftarrow \lambda \times \frac{3}{2}$
             16. $\gamma_2 \leftarrow 0$
          17. else
          18. $\lambda \leftarrow \lambda \times \frac{2}{3}$
          19. if $\lambda > \lambda_{max}$ then
             20. $\gamma_2 \leftarrow 0$
          21. end if
          22. end if
          23. end while
          24. $e_{v,1} \leftarrow \text{MINIBATCH}(D_v, p, 0)$
          25. if $e_{v,1} e_{v,1} > e_{v,0} e_{v,0}$ then $\triangleright$ Validation fails.
             26. $\gamma_1 \leftarrow 0$
          27. end if
          28. end while
      11. end while
   12. end for
13. function $[e, J]=\text{MINIBATCH}(D_{tr}, p, \delta)$
   14. for $s = 1$ to $\lceil D_{tr} \rceil$ do
   15. $y \leftarrow \Omega_L(u_s)$
   16. $e_s \leftarrow [e(1) \ldots e(T)]^T$ $\triangleright$ eq. (20)
   17. if $\delta = 1$ then
   18. $J_s \leftarrow \frac{\partial \Psi_s}{\partial p}$ $\triangleright$ eq. (22)
   19. end if
   20. end for
   21. $e^T \leftarrow [e^T \ldots e^T_{D_{tr}}]$ $\triangleright$ eq. (21)
   22. if $\delta = 1$ then
   23. $J^T \leftarrow [J^T_1 \ldots J^T_{D_{tr}}]$ $\triangleright$ eq. (22)
   24. end if
   25. end function

modelling are well established and can be found in, among others, [13]. For our simulation, the model to generate the altitude data is given by

$$\ddot{z} = \frac{1}{m} \left( k_t u^2 (1 + f_{ge}^2) - c_d \dot{z} - mg + \eta \right)$$

$$f_{ge} = k_{ge} \frac{h_{ge} - \min(h_{ge}, z)}{h_{ge}}.$$  

In equation (26), $z$ is the altitude of the vehicle and $u$ is the sum of all four motor speeds (in RPM), which relates directly to the thrust produced. The mass of the quadrotor is $m$, $k_t$ is the thrust co-efficient, $c_d$ is the drag coefficient and $f_{ge}$ is a simple model for ground effect. The ground effect acts at altitudes lower than $h_{ge}$, and $k_{ge}$ is the ground effect co-efficient. Finally, $\eta$ is a white noise. Table I lists the values used for data generation. To generate a more realistic dataset, we would like to have flyable trajectories, and as such, a random noise as the input to equation (26) is not considered. Instead, each altitude trajectory was created by applying a random sum of 10 sinusoids with variable frequencies in range of $[1, 10] Hz$ as input. To capture the ground effect, altitude is varied in the range $[0, 2]$ meters over the whole dataset. The dataset is collected using a $f_s = 10 Hz$ sampling frequency. In Fig. 3 one sample of the dataset with normalized values is illustrated.
accurately learn the quadrotor model with a MODERNN with 48 weights, computation times were significantly improved over both RMLP and NARX, to approximately 0.5 hours from 5.5 and 7.5, respectively.

An example of the magnitude of the condition number of the Hessian matrix during the training of the three networks is presented in Fig. 4. For the same $n_{tr}$, NARX and RMLP lead to higher condition numbers than MODERNN as the optimization progresses, which may contribute to failure of NARX and RMLP to learn on small minibatches. Since the only difference between MODERNN and RMLP is the presence of forward connections in MODERNN, both having the same number of layers and hidden neurons, we conclude that forward connections contribute to a less ill-conditioned Hessian.

### VI. CONCLUSION AND FUTURE WORKS

In this paper, we present a modular and flexible class of RNN architectures for modelling of high-order dynamic and nonlinear systems. MODERNN not only encapsulates a large class of already well-studied RNNs, but also provides a modular and easy-to-implement way of calculating network Jacobians. We also demonstrate that feedforward connections in a RNN can play a significant role in alleviating vanishing/exploding gradient in space. We provide a basic learning algorithm based on the LM optimization method, which illustrated successful application in learning altitude dynamic of a simulated quadrotor. By enabling smaller networks to learn the same complexity dynamics, we have dramatically reduced the necessary computation time to achieve reliable models for dynamical systems of moderate complexity with significant nonlinearities present.

### REFERENCES


