# Modelling Quadrotor Dynamics Using Neural Networks

Melissa Mozifian

*Abstract*— **This work explores the feasibility of deploying deep recurrent neural network architecture to model quadrotor dynamics by applying autoencoders as a pre-training technique. Deep neural networks have many levels of non-linearities, allowing them to compactly represent highly nonlinear functions. However this leads to a more complex architecture and the more complex the architecture, the higher capacity the network will have. Although more complex functions can still be estimated by the network, this also makes the network more vulnerable to overfitting. Furthermore, a complex network might be too sensitive to initial weight values. By introducing an autoencoder into the training process, we hope to introduce two factors: improve flexibility of the architecture we train on and better initial weight values. Based on our experiments, this provides a good trade off : the network learns more frequently as it is less sensitive to random initial weights and it is also less prone to overfitting.**

## I. INTRODUCTION

Over the last few years, quadrotors have attracted a lot of attention from the research community. Quadrotor control in particular is interesting due to the resulting dynamics of the system which is highly nonlinear. Although there have been quite a few successful attempts to capture the main dynamics of the quadrotor using physical properties of the system [1], [2], achieving a stabilized control yet remains a challenge. This is due to the nonlinear dynamic behavior of quadrotors, making it difficult to obtain an accurate mathematical model. Because quadrotors are dynamic systems, any approach to model their behavior must be able to capture both dynamic characteristics and nonlinearities of the model. Example of such nonlinearities in a quadrotor include aerodynamic friction, uncertainties such as mechanical impreciseness, sensor noise, different actuators, vibrations and other phenomena.

To improve a quadrotor model, one feasible approach is to model such nonlinearities using experimental data [3]. In particular, neural networks have been used in modelling and control of dynamic systems such as [4] [5] [6] and [7]. Among various architectures, Recurrent Neural Networks (RNNs) [8] have proven to be successful at learning nonlinear dynamic behavior due to the inclusion of feedback signals, giving them the ability to represent dynamical systems.

Although adding more layers to the RNN architecture can increase the capacity of the network to represent higher order and more complex nonlinearities, this is usually avoided due to the vanishing/exploding gradient [9] problem. More generally, the following challenges remain in training a deep network :

1) Curse of dimensionality : As the size of the input and output of the model increases as well as the complexity of the mapping, the optimization process becomes more challenging. This phenomenon is known as the curse of dimensionality.
2) Vanishing/Exploding gradient : During the gradient back-propagation, the gradient can end up being multiplied a large number of times resulting in the magnitude of gradients in the lower layers being much smaller than in higher layers. If the weights are small, it can lead to a situation called vanishing gradient, where the gradients get so small that learning either becomes slow or stops completely (in the case of a RNN architecture, this problem exists both in the time and space domains of an RNN). On the contrary if the weights are large, it can lead to gradients being too large and hence causing the learning to diverge. This is referred to as exploding gradients. One way to address these problems is by initializing the weights very carefully.
3) Hyper-parameter value selection : Too many parameter causes the network to memorize the training data and hence does not generalize well.

This work proceeds as follows. The next section briefly describes the background in dynamic motion modelling and the use of RNNs to model non-linear dynamic motion. Then we will study an existing RNN architecture, MODERNN [3] which enables constructing a recurrent neural network architecture. We then introduce techniques to apply an autoencoder to MODERNN to improve the training process. In section VI the simulation results of modelling the altitude of a quadrotor are presented and compared with varying the input of a MODERNN architecture to include history versus training an autoencoder to encode the input history as an alternative to the full history.

## II. LITERATURE SURVEY

### A. System Classifications

*Static vs Dynamic:* We define a system as a collection of elements that interacts with its environment via a set of input variables $u$ and output variables $y$, where the variable could be a scalar or a vector. A system can be classified in different ways, for instance it can be a continuous time system where the system evolves with time indices $t \in \mathbb{R}$, as opposed to a discrete time system with time indices $t \in \mathbb{Z} = \{..., -3, -2, -1, 0, 1, 2, ...\}$. Examples of a continuous time system is pendulum and a discrete time system is economic activity data, where it is possible to measure economic activity discretely [10]. Furthermore, a system can be static or dynamic. A system is classified as static if its output depends only on its present input. We can represent a static system by a function $f(u,t)$ such that for all $t \in T$ ,

$$y(t) = f(u(t), t). \tag{1}$$

In contrast, a dynamic system requires past input to determine the system output. So to determine $y(t)$ we need to know $u(\tau), \tau \in (-\infty, t]$. We can think of dynamical systems as having a real valued hidden state that cannot be observed directly. In order to predict the output we need to infer the hidden state.

*Linear vs Non-linear:* A linear dynamical system's evaluation functions are linear. In a non-linear system the output is not directly proportional to the input. Typically we use mathematical approaches to find an approximate system to represent the behavior of a nonlinear system.

### B. Quadrotor Dynamics modelling

For this work, a simulator written in Matlab was used to generate the training dataset for modelling the altitude of a simulated quadrotor. The simulator is based on a physical model of a quadrotor which is described in detail in [11] and [12]. The training dataset consists of 5001 (before downsampling) input-output pairs representing one flight, with 500 flights in total. The input only consists of motor inputs over time sequence with the same length for each flight. The output trajectory is the desired altitude of the quadrotor at each time instance. It is indeed possible to learn the full quadrotor dynamics by having more than one output, however for simplicity and to reduce training time, we chose to learn only altitude as output as less effort is required in computing the gradients.

## III. Autoencoders

Autoencoders belong to the class of unsupervised learning algorithms. Autoencoders were first introduced in the 1980s by Hinton [13]. The autoencoder tries to learn the function $h_{W,b}(x) \approx x$ which can be regarded as learning an approximation to the identity function. By placing constraints on the network, such as limiting the number of hidden units, the network can learn interesting structure about the data. The hidden layer $h$ in the network describes a code used to represent the input. If an autoencoder succeeds in simply learning to set $g(f(x)) = x$ everywhere, it will not be useful. Instead, they are designed to apply restrictions to the network, allowing them to produce an output that resembles the training data.

More recently, autoencoders have been used in the deep architectures such as Deep Belief Networks [14] to pretrain the network. This approach is based on the observation that pre-training each layer with an unsupervised learning algorithm, yields a better starting point with respect to the initial weights.

### A. Formal Definition

Formally, an autoencoder takes an input $x \in [0, 1]^d$ and encodes it to a hidden representation $y \in [0, 1]^{d'}$ through a deterministic mapping function :

$$y = s(Wx + b) \tag{2}$$

Where $s$ is an activation function, such as a sigmoid function. The resulting $y$ i.e. the code is then decoded back into a reconstruction $z$ of the same shape as $x$ through a similar transformation :

$$z = s(W'y + b') \tag{3}$$

We can think of $z$ as prediction of $x$ given the code $y$. Depending on the appropriate distributional assumption about the input, we can choose a method to measure the reconstruction error. One common approach is using the squared error :

$$L(x, z) = ||x - z||^2 \tag{4}$$

The hope is that the code $y$ yields a distributed representation, capable of capturing the main variations in the data.

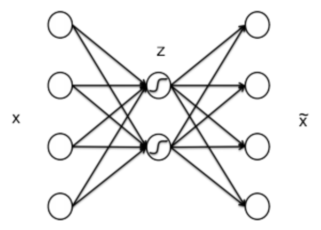

Fig. 1. An autoencoder learning a "compressed" representation of $x$.

### B. Feature Detectors Learnt from Image Patches

In a regular autoencoder, the encoded representations are constrained by the size of the hidden layer. So we can decrease the dimension of the input by having less number of neurons than the input features. Training a neural network that produces an output that is identical to the input with fewer nodes in the hidden layer, resembles a compression algorithm. In which case, the redundant information is thrown away and only the important features are kept.

We could also increase the input dimensionality by having more neurons than our input features. Another way to constrain the representation is by introducing additional term to our cost function such as a sparsity constraint. Ideally we want fewer units to fire at a given time. To get an intuition of what an autoencoder learns, we trained a sparse autoencoder on 8 x 8 patches from whitened natural image data.

We formulate a sparse autoencoder as follows. Suppose $a^2{}_j(x)$ denotes the activation of the hidden unit j when the network is given a particular input $x$. Further, let :

$$\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m}[a_j^{(2)}(x^{(i)})] \tag{5}$$

be the average activation of hidden unit $j$. A sparse autoencoder enforces the following constraint:

$$\hat{\rho}_\mathrm{j} = \rho \tag{6}$$

where $\rho$ is a sparsity parameter, typically a very small value like 0.05. This brings the average activation of each hidden neuron $j$ to be closer to $\rho$ and to satisfy this constraint, the hidden unit's activation must then mostly be near 0.

To achieve this goal, we add an extra penalty term to the optimization objective that penalizes $\hat{\rho}_\mathrm{j}$ and causing it to deviate from. A popular sparsity constraint is based on the Kullback-Leibler divergence:

$$KL(\rho||\hat{\rho}_j) = \rho log\frac{\rho}{\hat{\rho}_j} + (1-\rho)log\frac{1-\rho}{1-\hat{\rho}_j} \tag{7}$$

The Kullback-Leibler divergence is a measure of the information lost when probability distribution $\hat{\rho}_\mathrm{j}$ is used to approximate $\rho$. The overall cost function then becomes :

$$J_{sparse}(W,b) = J(W,b) + \beta\sum_{j=1}^{s_2} KL(\rho||\hat{\rho}_j) \tag{8}$$

where J(W,b) is the cost function and $\beta$ controls the weight of the sparsity penalty term.

*Implementation Notes:* The autoencoder is essentially a Multi-layer Perceptron with one hidden layer. We can train a neural network using gradient descent optimization algorithms and obtaining gradient descent updates requires computing the partial derivatives of the weight matrices and bias vectors. We use backpropagation algorithm which is an efficient way to compute these partial derivatives. Once we have defined our cost function $J(\theta)$ and computed the gradients, we can use any optimization algorithm, perhaps more sophisticated than gradient descent to minimize $J(\theta)$. For instance some algorithms automatically tune the learning rate $\alpha$ and try to determine an appropriate stepsize to reach local optimum as quickly as possible. One example is L-BFGS optimization algorithm [15] which we have used for our autoencoder. Finally when implementing the backpropagation algorithm, it is a good idea to numerically check the derivatives to ensure the implementation is correct.

The network was configured with 64 input units (since each input vector consists of a $8x8$ image patch), 25 hidden units, and 64 output unit, same dimension as the input. The initial weights and biases were initialized to random numbers drawn uniformly from the interval $[-\sqrt{\frac{6}{n_{in}+n_{out}+1}}, \sqrt{\frac{6}{n_{in}+n_{out}+1}}]$, where $n_{in}$ is the number of inputs feeding into each node, and $n_{out}$ is the number of units that a node feeds into. The following parameters were set as follows : $\lambda = 0.0001$ , $\beta = 3$ and $\rho = 0.01$, where $\lambda$ is the weight decay (which tends to decrease the magnitude of the weights and helps overfitting), $\beta$ is the sparsity penalty term and $\rho$ is the sparsity parameter corresponding to the average activation as explained above.

Figure 2 aims to give an intuition of what a sparse autoencoder has learned. We can see how the weights of an autoencoder corresponds to underlying features which are edges in this case.

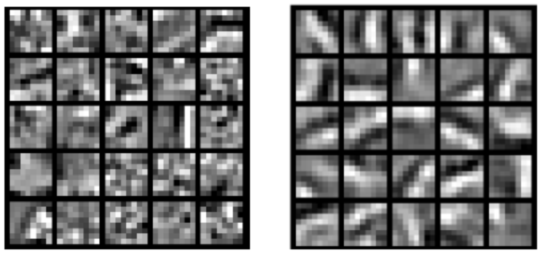

Fig. 2. First image to the left displays randomly selected image of 8 x 8 patches from the input pool of 10 images, and the second image shows edge features learned by the autoencoder.

## IV. Recurrent Neural Network

RNNs are dynamical models with powerful representational capabilities. They are inspired by the property of persistence memory in the brain. One limitation of vanilla neural networks is that they are not able to reason about previous events in order to predict future events. RNNs address this issue by combining their input vector with their state vector to produce a new state vector. The resulting output vector's content is influenced not only by the given input but also on the entire history of inputs [16].

Figure 3 demonstrates the concept of RNN allowing information to persist by enabling a feedback loop. More generally, the properties that make RNNs powerful are :

- Hidden states allowing information to be stored about the past in an efficient manner
- Non-linear dynamics enabling update of hidden states in complicated manner.

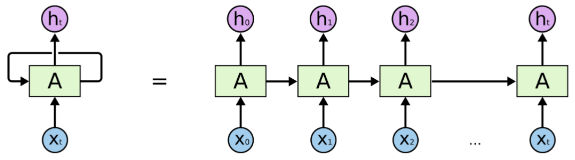

Fig. 3. An unrolled recurrent neural network.

### A. Modular Deep Recurrent Neural Network (MODERNN)

MODERNN [17] is a general form of a structurally deep RNN. This architecture is a generalization over Recurrent MultiLayer Perceptron (RMLP) with locally recurrent layers $(G_i)$ with $m_i$ inputs and $n_i$ outputs and sequentially connected i.e. each layer is connected to itself and the next layer. A simple MODERNN with three layers is shown in Figure 4. The equations governing the dynamic of $G_i$ are as follows :

$$\begin{aligned} x_i(k) &= A_i y_i(k-1) + B_i u_i(k) + b_i \\ y_i(k) &= f_i(x_i(k)), \end{aligned} \tag{9}$$

where $k$ is the discrete-time index, and :

- $x_i(k) \in \mathrm{I\!R}^{\mathrm{n_i}}$ is the state of the layer,

- $y_i(k) \in \mathrm{I\!R}^{\mathrm{n_i}}$ is the output layer,
- $u_i(k) \in \mathrm{I\!R}^{\mathrm{m_i}}$ is the input to the layer,
- $A_i \in \mathrm{I\!R}^{\mathrm{n_i}}$ x $\mathrm{I\!R}^{\mathrm{n_i}}$ is the feedback weight matrix,
- $B_i \in \mathrm{I\!R}^{\mathrm{n_i}}$ x $\mathrm{I\!R}^{\mathrm{m_i}}$ is the input weight matrix,
- $b_i \in \mathrm{I\!R}^{\mathrm{n_i}}$ is a bias weight vector,
- $f_i(.)$ 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.

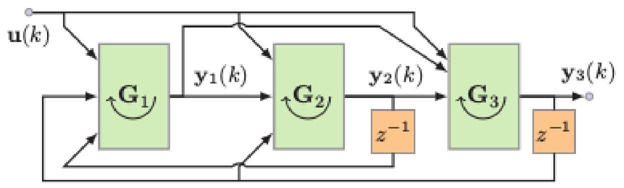

Fig. 4. A 3 layer MODERNN architecture with output $y_3(k)$.

We will refer to $x_i(k)$ and $y_i(k)$ as the network states as long as $y_i(k)$ is getting fed back into the system and does not form the resulting output(s). More detail of the internal architecture of the network is described in [17].

## B. Learning Algorithm for training MODERN

For modelling the quadrotor's altitude dynamics, each training sample consists of a motor input (sum of motor speeds) and a position output (altitude of the vehicle), both over time. Note that taking the sum of motor speeds, simplifies the model. The network adopts a batch learning method using Adadelta [18] as the optimization algorithm. The advantage of Adadelta is that we do not need to set a default learning rate. The mini-batch size is set as a hyperparameter. Note that the mini-batch trains the training example in batches. As a result it minimizes the gradient matrix and hence consumes less memory. Usually by increasing the mini-batch size we get a better and faster convergence, in terms of the iterations. Although this leads to each iteration taking much large amount of time and memory.

## V. Pre-training a RNN with an Autoencoder

The main objective of this work was to explore techniques to pre-train a RNN. Hence as a starting point, a regular autoencoder was used to reduce dimensionality of the input. The time-series data we used to train the network are trajectories in time. In a typical training scenario, only one step of a time-series data is given to the network at time $t$.

Note that the motivation is the ability to connect previous information to the present input efficiently and effectively. Although in our problem, we do not have a long-term dependency (for instance consider word prediction where there exists a long-term dependency), it is still useful to allow more information to flow into the network. One might expect that if we provide a history of previous time-steps along with the input to the network, it would generalize better. This is based on the observation that the network has more information at hand. However increasing the dimension of input is computationally more expensive and causes the learning to slow down significantly as well as introducing additional complexity to the network which in turn causes overfitting. For that reason, we deploy an autoencoder that given an input history with certain step size, we refer to this as the number of steps throughout the history, it maps the history to an encoded output.

We preprocess the time-series input as follows : Let's assume we have a time-series signal. Now in order to process this input for the autoencoder, we are going to take snap shots of this data, using a $step\_size = 5$. We then proceed using a sliding window analysis on the signal. We capture 5 time steps in the sequence, and slide the window by one step and capture the next sequence of length 5. We repeat this for the entire data. As a result, we end up with a 2D matrix representing the repeating pattern in our signal. This matrix is then fed to the autoencoder. This extracted sequence, corresponds to the entire history (of 5 time steps) at time $t-1$. We map this sequence history to the input at time $t$. In our case since we are using only one neuron in the hidden layer, to learn our sequence history, we obtain one activation.

More formally assume we have the following segment of the time-series data :

$(s1, t1)$,$(s2, t2)$,$(s3, t3)$,$(s4, t4)$,$(s5, t5)$,$(s6, t6)$

Now the activation of the encoded history of $s1 - s5$ is mapped with current input $s6$. We repeat this by taking one step at a time and capturing the history for each input. Note that clearly there is no history for the first 5 steps.

The autoencoder was trained using Adadelta as the optimization algorithm and Mean Squared Error (MSE) as the objective function, with number of epochs set to 200 and mini-batch size set to 10. We used a sigmoid activation function for hidden layer (encoder) and linear activation function for the output layer (decoder).

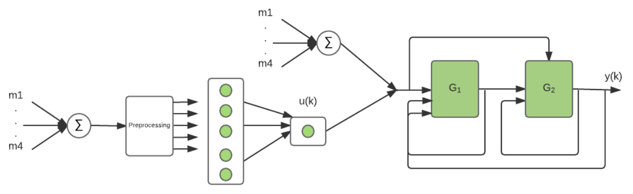

Fig. 5. Architecture used for this work consisting of a 2 layer MODERNN network with two sets of inputs :one input at step time $k$ and another history sequence of all previous inputs encoded by the autoencoder.

For this work, we adopted the architecture demonstrated in Figure 5, where the output of an autoencoder is fed back into the network. Note that in this case, the output of the autoencoder refers to the activation of the hidden layer neurons, and in this case there is only one neuron. Since we are using the autoencoder's activation as input, we do not need to decode the output. Although the decoding component can be used to ensure the correctness of the autoencoder. That is, the reconstructed output of an autoencoder must be a close approximation of the original input. Figure 7 demonstrates an example of original sum of motor speed data and the reconstructed version using an autoencoder.

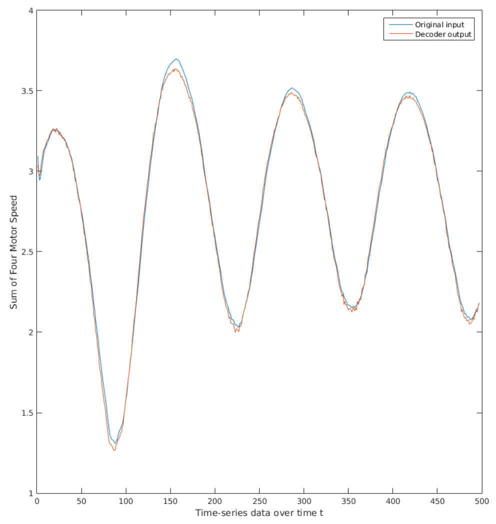

Fig. 6. This figure demonstrates the reconstructed output from an autoencoder showing how close the reconstructed output is to the original input.

*Network Initialization*

Due to the recursive nature of RNNs, it is necessary to initialize the initial activity state for all the hidden and output units. This is because the evolution of the network states is not known a priori and highly dependent on the network weights. It is better instead to treat the initial states as a learned parameter. We start off by initial random states, at the end of each training sequence, errors are backpropagated through time and initial states are updated. In the case of MODERNN, a separate network is trained and the output is used as the initial states as described in [17].

## VI. Simulation Results

We apply the MODERNN structure to the problem of learning quadrotor altitude dynamics. The details of quadrotor modelling can be found in [1]. For the simulated data, the model to generate the altitude is given by:

$$\begin{aligned} \ddot{z} &= \frac{1}{m}(k_t u^2(1+f_{ge}^2) - c_d\dot{z} - mg + \eta) \\ f_{ge} &= k_{ge}\frac{h_{ge} - min(h_{ge}, z)}{h_{ge}}. \end{aligned} \tag{10}$$

In equation 10, $z$ is the altitude of the vehicle and $u$ is the sum of all four motor speeds which relates directly to the thrust produced. The mass of the vehicle 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 altitude lower than $h_{ge}$, and $k_{ge}$ is the ground effect coefficient. Finally $\eta$ is a white noise. Table VI shows the values used for data generation. The first architecture, SISO, refers to single-input-single-output, HISO, history-input-single-output, EISO, encoded-input,single-output.

The dataset is collected using a $f_s = 10Hz$ sampling frequency.

TABLE I
Quadrotor parameters used in the simulation.

| m | $k_{ge}$ | $h_{ge}$ | $k_t$ | $z_{max}$ |
|---|---|---|---|---|
| 1 kg | 0.5 | 1 m | $1.95x10^-5\frac{Ns^2}{rad}$ | 2 m |

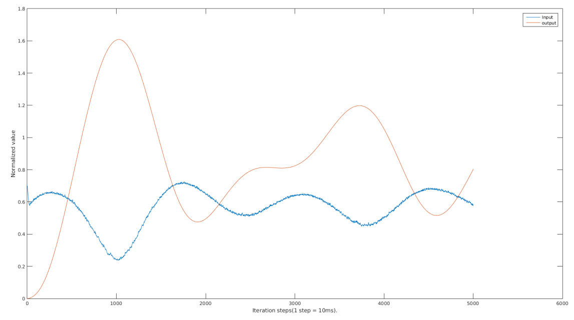

Fig. 7. A generated data sample.

## VII. Empirical Comparison of Deep Network Training Strategies

For the experiments, the parameters used are: the number of layers $L$, the number of hidden neurons in each layer $h$, the size of the mini-batch $n_{tr}$ and $q$ is the number of weights used throughout the network. Note that we chose a setup with the same number of layers and neurons with varying mini-batch size. The results are summarized in Table VII. In this table, $E_m$ is the mean value of the errors over the entire dataset and $it_{tr}$ is the minimum training iteration at which the best solution was found. The result is the averaged error over training 10 networks.

Each mini-batch is divided into two small sets: training set and validation set. For the experiments the split of training and validation is 70% and 30%. By performing a cross-validation test, the algorithm prevents overfitting to each mini-batch. However when the network is too rich, it could overfit on both the training and the validation part of the mini-batch. Hence this causes instability from one mini-batch to another (and this indeed occurs in our experiments as explained later.).

### A. Results Interpretation

Based on the mean value of the errors, it is clear the EISO network learned more often. We can observe this in Figure 8 since the average error for the input with the encoded history is lower. Although one might expect feeding the entire history should perform better, this was not always the case as the issue of overfitting arose specially in cases where the mini-batch size is small.

In the case where the mini-batch is larger, the HISO network performs a lot better although EISO network still outperforms HISO. It is also surprising to see the EISO network to perform worse with higher mini-batch size, however one explanation could be that the number of experiments are not large enough to reflect a good average. This does

TABLE II
EXPERIMENT RESULTS.

| Network | $L$ | $h$ | $q$ | $n_{tr}$ | $E_m$ | $it_{tr}(h)$ |
|---|---|---|---|---|---|---|
| SISO | 2 | 5 | 48 | 10 | 111.3205 | 7 |
| HISO | 2 | 5 | 78 | 10 | 128.2081 | 18 |
| | 2 | 5 | 78 | 20 | 88.8497 | 8 |
| EISO | 2 | 5 | 54 | 10 | 32.1858 | 7 |
| | 2 | 5 | 54 | 20 | 58.0377 | 12 |

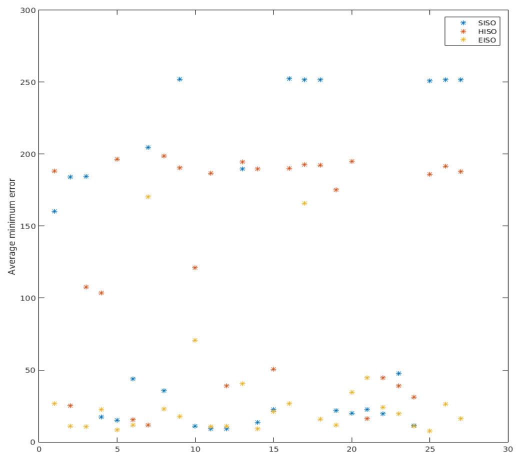

Fig. 8. Average training error of 3 sets of experiments involving 10 networks with random initial weights.

occur since the initial weights are uniformly random and we are only running 10 small networks. Note that we do expect the HISO to perform better by increasing the mini-batch size, although this increases the training time but in turn it leads to less overfitting. Overall the results show the flexibility of EISO over HISO and demonstrates the trade-off between the complexity and overfitting. Another thing to note is the number of weights in the network. Note that the HISO network has the largest number of weights and this leads to a more complex network.

Based on this result, we can conclude that EISO on average learns more often than SISO and does not suffer from overfitting to the degree that occurs in HISO.

## VIII. CHALLENGES

Although the main objective of this project was to deploy autoencoders as means of weight initialization to pretrain a RNN architecture, this task proved to be challenging. The fact that input to a RNN is time-series as opposed to static, makes this task difficult in order to achieve more reasonable initial weights. Hence an effective pre-training technique for the purpose of weight initialization remains an open question. Therefore we decided to explore applying other pre-training techniques using autoencoders such as input dimensionality reduction.

The initial autoencoder implemented in Matlab lacked the flexibility of parameter tuning as well as experimenting with various optimization algorithms to select the best performance. Hence we resorted to using Keras library [19] to implement an autoecoder to achieve better results and performance and hence speed up the training process.

Another challenge was the MODERNN long training times, as running one experiment for the full dynamic model takes at least a few days. And hence for simplicity and reducing the training time, the model was further simplified and only the motor speeds were used as input. This enabled us in turn to take the average of 10 trained network under three different settings with varying mini-batch sizes. We train over more than one network in order for the results to reflect a good average. We chose this to demonstrate the performance of the network in the default mode i.e. taking one time-step input at a time, versus feeding the network with the history versus the encoded history. We allocated at least a week for running experiments, although this was still not sufficient. Specially after increasing the mini-batch size, the training time increased. Unfortunately the SISO results with mini-batch size 20 were lost due to power cut to one of the machines. Although we suspect the results to be an improvement over the smaller mini-batch size.

## IX. CONCLUSIONS

This work explored pre-training techniques for a particular RNN architecture so-called MODERNN. Using a sparse autoencoder we showed how an autoencoder is capable of learning underlying features of an image input and we visually demonstrated this. We then applied an autoencoder to time-series data and showed that an autoencoder is still able to extract interesting structure about the data.

The network was trained in three settings, one with one input and one output fashion, one with the full history of input and one with an encoded history of input and the input itself. This work demonstrated the effectiveness of applying autoencoders to "compress" the input. An autoencoder was trained separately and the output activations were fed to the MODERNN network. We demonstrated that the encoded input version learned more often in average and converged in less iterations. We also demonstrated the network with the full history used large number of weights and also lead to overfitting. And therefore in order to introduce more information to the network, using an autoencoder to reduce input of dimension leads to the network learning useful features. This is because the network trained with an autoencoder was not too complex to frequently overfit on small mini-batches, compared to the more complex network i.e. network with the full input history. And hence autoencoder provides a nice balance between introducing additional information and the avoidance of defining a highly complex function. In this work, it is shown that using an autoencoder to encoded input history, it is possible to successfully train a deep recurrent network on a long time series.

## X. Future Work

Future work will apply input dimensionality reduction to learn the full dynamic model of a quadrotor as well as training on real data as opposed to simulated data.

Another interesting technique is applying autoencoders to both input and output. For instance we can introduce delay in the network output prediction, where the predictions are accumulated and an autoencoder is trained to compress the output and only feed one encoded output back into the network. This in theory reduces burden of introducing additional weights which must be fed back to all the previous layers, just as we applied to the input.

Finally another interesting work might be experimenting with autoencoder variations such as sparse or deep autoencoders where multiple levels of encoding is applied.

## ACKNOWLEDGMENT

We would like to thank the author of MODERNN [3] for giving us access to their framework and providing guidance on training the network.

## APPENDIX

The code for this work has been included along the submission and the code repository can be found on : https://github.com/mfm6/Modelling-Quadrotor-Dynamics Details on organization of the code can be found in the README.md file. Since the MODERNN framework is a private repository, access will be provided upon request in order to reproduce the experiments presented in this work.

## References

[1] G. M. Hoffmann, H. Huang, S. L. Waslander, and C. J. Tomlin, "Precision flight control for a multi-vehicle quadrotor helicopter testbed," *Control Engineering Practice*, vol. 19, pp. 1023–1036, September 2011.

[2] A. Das, F. Lewis, and K. Subbarao, "Backstepping approach for controlling a quadrotor using lagrange form dynamics," *Journal of Intelligent and Robotic Systems*, vol. 56, no. 1, pp. 127–151, 2009.

[3] N. Mohajerin and S. L. Waslander, "Modelling a quadrotor vehicle using a modular deep recurrent neural network," in *Systems, Man, and Cybernetics (SMC), 2015 IEEE International Conference on*, pp. 376–381, Oct 2015.

[4] L. Krl and M. imandl, "Predictive dual control for nonlinear stochastic systems modelled by neural networks," in *Control Automation (MED), 2011 19th Mediterranean Conference on*, pp. 1277–1282, June 2011.

[5] O. Nerrand, P. Roussel-Ragot, D. Urbani, L. Personnaz, and G. Dreyfus, "Training recurrent neural networks: why and how? an illustration in dynamical process modeling," *IEEE Transactions on Neural Networks*, vol. 5, pp. 178–184, Mar 1994.

[6] I. S. Baruch and C. R. Mariaca-Gaspar, "A levenberg&ndash;marquardt learning applied for recurrent neural identification and control of a wastewater treatment bioprocess," *Int. J. Intell. Syst.*, vol. 24, pp. 1094–1114, Nov. 2009.

[7] A. Delgado, C. Kambhampati, and K. Warwick, "Dynamic recurrent neural network for system identification and control," *IEE Proceedings - Control Theory and Applications*, vol. 142, pp. 307–314, Jul 1995.

[8] G. E. W. R. J. Rumelhart, David E.; Hinton, "Learning representations by back-propagating errors," pp. 533–536, 1986.

[9] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *Trans. Neur. Netw.*, vol. 5, pp. 157–166, Mar. 1994.

[10] P. P. Y. Li, "Me8281 advanced control system design." University Lecture, 2016.

[11] J. Ghandour, S. Aberkane, and J.-C. Ponsart, "Feedback linearization approach for standard and fault tolerant control: Application to a quadrotor uav testbed," *Journal of Physics: Conference Series*, vol. 570, no. 8, p. 082003, 2014.

[12] G. M. Hoffmann, H. Huang, S. L. Waslander, and C. J. Tomlin, "Precision flight control for a multi-vehicle quadrotor helicopter testbed," *Control engineering practice*, vol. 19, no. 9, pp. 1023–1036, 2011.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1," ch. Learning Internal Representations by Error Propagation, pp. 318–362, Cambridge, MA, USA: MIT Press, 1986.

[14] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, pp. 1527–1554, July 2006.

[15] M. Schmidt, "minfunc: unconstrained differentiable multivariate optimization in matlab." Library, 2005.

[16] C. Olah, "Understanding LSTM networks." Tutorial Blog, 2015.

[17] N. Mohajerin and S. L. Waslander, "Modular deep recurrent neural network: Application to quadrotors," in *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 1374–1379, Oct 2014.

[18] M. D. Zeiler, "Adadelta: An adaptive learning rate method," *CoRR*, vol. abs/1212.5701, 2012.

[19] Keras, "Deep learning library for theano and tensorflow." Accessed: 2016-08-04.