Empirical Evaluation of Recurrent Neural Networks for System Identification

Nolan Wagener
Interactive Computing
Georgia Institute of Technology
Atlanta, GA 30332
nolan.wagener@cc.gatech.edu

Abstract

In this paper, we introduce a method to train RNNs to perform system identification of a physical system. Such a method includes a pretraining scheme that allows the RNN to learn the state representation, state dynamics, and observation equation in separate steps. We train RNNs both with and without this pretraining scheme along with a baseline linear system from N4SID–EM and compare their performance. We find that pretraining the RNN doesn't hurt performance and for the most part makes it easier to train the network. However, learned linear systems perform better on simpler systems (like SDOF mass-spring-dampers), with them only performing worse than RNNs on the MDOF mass-spring-damper.

1 Introduction

System identification is a deeply studied field that has widely varying applications in areas like control, machine learning, and finance. A classic approach within this field is to fit some parameters of a given model using least squares or maximum likelihood techniques [10]. However, the model must be relatively accurate and all necessary features must be measurable.

Of course, a system of interest may have difficult-to-model effects such as friction, backlash, and deadbands for mechanical systems. Furthermore, a system may not be fully observable and so unobserved states would have to be estimated. A way to combat both of these problems is through latent space approaches (like subspace identification) and non-parametric methods.

Recurrent neural nets (RNNs) have potential for system identification and have achieved state-of-the-art success in problems such as handwriting and speech recognition. Within handwriting recognition, long short-term memory nets (LSTMs) have achieved start-of-the-art success by outperforming HMM methods by over 10% [6]. Within speech recognition, these networks outperformed the previous state-of-the-art technique by 0.6% [7]. There is also theoretical backing for using RNNs since, under mild assumptions, they have been shown to be universal approximators of general dynamical systems [3]. Finally, because they are parametric, they don't suffer from the same computational complexity problems that non-parametric methods do, since their complexity increases polynomially with the amount of data.

In this paper, we will empirically investigate how well RNNs perform for system identification in terms of accuracy, comparing them to linear systems generated from a linear subspace identification technique.

2 Related Work

One way of performing system identification is through subspace techniques. For instance, algorithms based on N4SID (numerical algorithms for subspace state space system identification) find a linear description for a partially observed system by performing a singular value decomposition of an oblique projection formed from the input and output data [13]. Other algorithms may be based on MOESP (multivariable output-error state space), which uses the LQ and singular value decompositions based on a matrix formed from the input, output, and user-defined weights [13]. The EM algorithm can then be used after either of the previous algorithms to further refine the system matrices it was given [4].

Non-parametric techniques such as Gaussian processes can also be used since they are not restricted to fitting to a user-given model. For instance, they have been used for learning dynamics and observations of a Bayes' filter to great effect when compared to parametric techniques [8].

RNNs have the capability of modeling dynamical systems with neural networks. For instance, there has been work in learning a deep recurrent model for fruits for the purposes of cutting [9]. The model was designed so that it would learn both short-term and long-term latent features, which would aid in encoding material properties. This RNN combined with model predictive control has allowed a PR2 to significantly outperform trajectory-based stiffness controllers for fruit cutting. Another neural network application came with modeling a helicopter [11]. Rectified linear units were used for the activation functions and were able to divide the data into different regions that corresponded to different portions of demonstrated trajectories. This method was able to achieve a 60% improvement in RMS acceleration error over linear least squares methods based that were on a "Linear Acceleration Model" (proposed in [1]) and its variants. Furthermore, validation error was noted to continually decrease as the number of hidden units increased, albeit with diminishing returns. Two things of note are that the network was feedforward instead of recurrent and the predictions were of the linear and angular accelerations of the helicopter and not of the next state.

3 Model

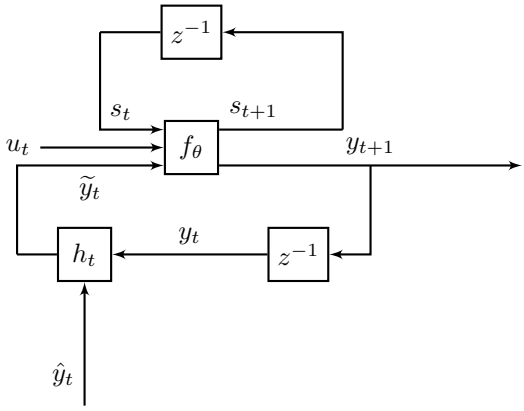


Figure 1: Our model

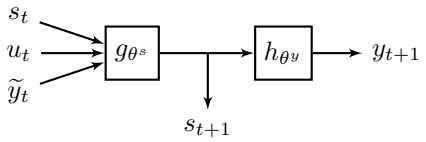


Figure 2: Our implementation of f_θ

Our model is based on the block diagram shown in Figure 1. Here, f_θ is an RNN that is parameterized by some weights θ and has a state s_t. The external input to the RNN is given by u_t and an

observation $\widetilde{y}_t$ (which is either the true observation $\hat{y}_t$ or the predicted observation y_t) is also fed into the network. When $\widetilde{y}_t = \hat{y}_t$, the RNN will have its state corrected (much like how a Kalman filter will use the observation to correct a linear system state), and when $\widetilde{y}_t = y_t$, the RNN will infer the next observation. The latter may be useful in applications like model predictive control (MPC), where the model must evolve in open-loop for an extended period of time.

For our system, the RNN is a two-layer network (one layer for the state update and one layer for the observation) with the hyperbolic tangent function as the activation. The state update equation is

$$s_{t+1} = g_{\theta^s}(s_t, u_t, \widetilde{y}_t) = \tanh(W_s^s s_t + W_u^s u_t + W_y^s \widetilde{y}_t + b^s) \tag{1}$$

and is shown graphically in Figure 3. The observation equation is

$$y_{t+1} = h_{\theta^y}(s_{t+1}) = C \tanh(W_s^y s_{t+1} + b^y) + d \tag{2}$$

and is shown graphically in Figure 4. In total, our set of parameters for the RNN is $\theta = \{W_s^s, W_u^s, W_y^s, W_s^y, b^s, b^y, C, d\}$.

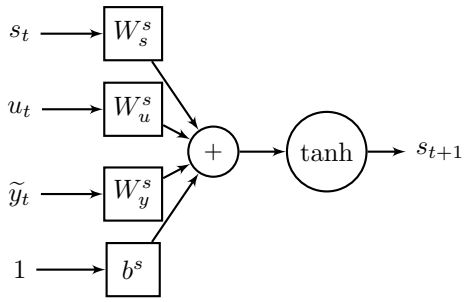


Figure 3: Our implementation of g_{θ^s}

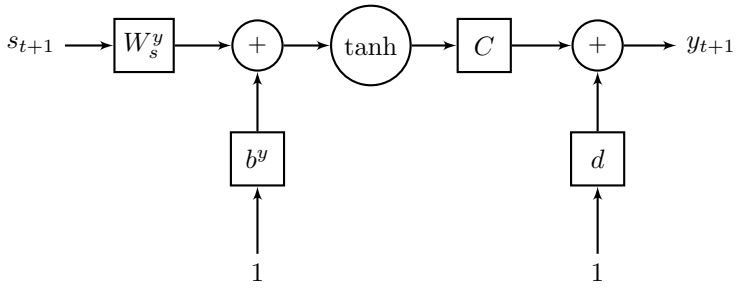


Figure 4: Our implementation of h_{θ^y}

The network f_θ itself will use those two inputs and its latent state to yield the next state and predicted observation (i.e. $(s_{t+1}, y_{t+1}) = f_\theta(s_t, u_t, \widetilde{y}_t)$).

4 Training Algorithm

Training is done using backpropagation through time (BPTT) with a sequence length of 50 [5]. For the gradient descent algorithm, we use rmsprop as it should help prevent the algorithm from thrashing about a local minimum [12]. During training, the value of $\widetilde{y}_t$ is chosen by a stochastic function h_t, which is defined as

$$h_t(y_t, \hat{y}_t) = \begin{cases} \hat{y}_t, & \text{with probability } \varepsilon_t \\ y_t, & \text{with probability } 1 - \varepsilon_t \end{cases}$$

The curriculum scheudle ε_t is chosen so that it will decrease from one to zero as t increases. This means that early on, the network will mostly be trained with the true observation, whereas towards the end it will mostly be trained on its own predicted observations. Being fed the true observations early will allow the network to more quickly set its parameters to do well on filtering. Training on

predicted observations will let the model become robust to its own mistakes and make inference accurate [2]. The schedule we decided to use was the inverse sigmoid decay[1], an example of which is shown in Figure 5. During validation, the network will be fed with the true observations $\hat{y}_t$ (i.e., do filtering) on the first half of the dataset and fed with the predicted observations y_t (i.e., do inference) on the second half.

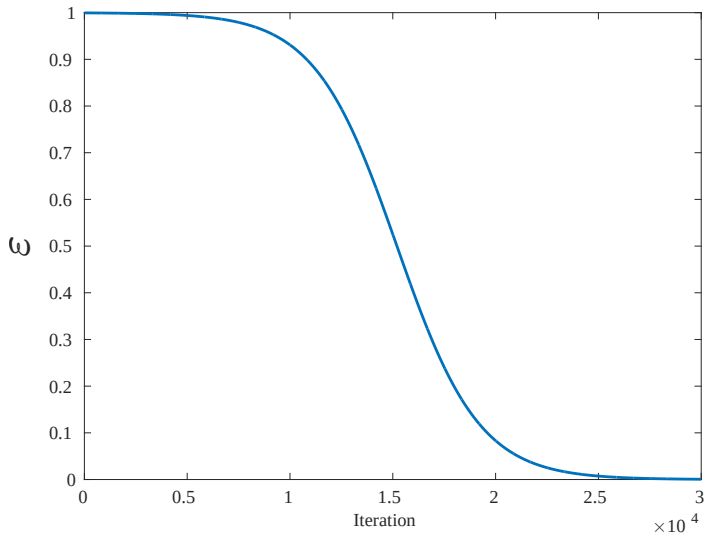


Figure 5: Inverse sigmoid decay schedule ($k = 2000$)

We also propose a pretraining scheme to help the RNN learn what the state representation, the dynamics of the state, and the observation equation should be in separate supervised regression steps.

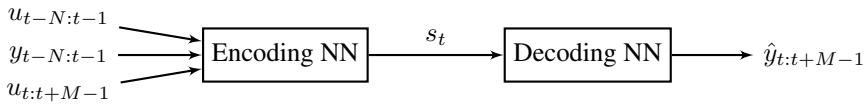


Figure 6: Pretraining step for learning state representation

For the first step, we propose finding a good state representation by compressing the past data and future inputs into a low-dimensional vector that can be used to predict the future observations. This low-dimensional vector will be considered our state. In order to do that, we learn both an encoding and a decoding neural network so that the predicted observations are close to the actual observations from the dataset. This function is shown in Figure 6. The input for this pretraining step is the dataset of inputs and observations, and the output is the collection of states $\{s_t\}$.

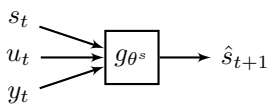


Figure 7: Pretraining step for learning dynamics

Next, we fit the dynamics in Equation (1) by making the predicted next state $\hat{s}_{t+1}$ close to the corresponding state from our dataset. The input to this pretraining step is the dataset of inputs and observations and the generated states from the previous step, and the output is the set of parameters $\theta^s = \{W_s^s, W_u^s, W_y^s, b^s\}$.

[1] $\varepsilon_t = k/(k + \exp(t/k))$, where $k \geq 1$

Figure 8: Pretraining step for learning observation

Finally, we fit the observation in Equation (2) by making the predicted observation $\hat{y}_t$ close to the corresponding observation from our dataset. The input to this pretraining step is the collection of generated states from the first pretraining step, and the output is the set of parameters $\theta^y = \{W_s^y, C, b^y, d\}$.

The RNNs parameters θ are then initialized from θ^s and θ^y and training through BPTT commences.

5 Results

Our code was implemented in Theano for its ease of use in prototyping neural network architectures and its speed of executing code on GPUs. All experiments were performed on an Intel Core i7 CPU and an NVIDIA Tesla K40 GPU. Because of the way CUDA functions were implemented in Theano, only single precision support was available.

To evaluate how well recurrent neural networks (RNNs) perform for system identification, we used their performance on some datasets we generated as a yardstick. These datasets came from systems of increasing complexity: a linear mass-spring damper system, a mass-spring damper system with a nonlinear spring, and a multi-degree of freedom (MDOF) mass-spring-damper with multiple nonlinearities. For the first two systems, the input is force and output is mass position. The third system comes from MATLAB's System Identification Toolbox. Figure 9 shows the model[2]. The input is motor torque and the output is the motor's angular velocity. For all datasets, Laplacian noise is added to the observations. For training and evaluation, we whiten the inputs and observations.

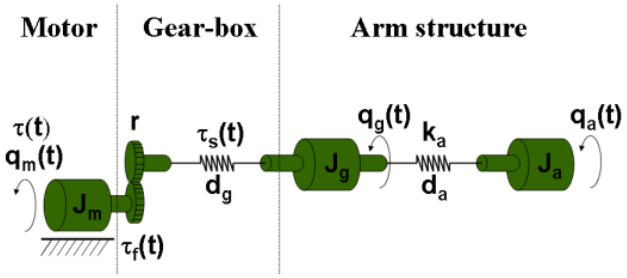


Figure 9: MDOF nonlinear mass-spring-damper

As baselines, we also trained RNNs without the pretraining scheme and generated linear systems from an N4SID–EM algorithm. For evaluation of the linear system, we use a trained Kalman filter with the linear system on the first half of the dataset, and allow the linear system to evolve in open-loop on the second half of the dataset.

The relevant figures and tables are:

- Numerical results: Tables 1, 2, 3
- Sample plots of predictions: Figures 10a, 11a, 12a
- Validation set loss: Figures 10b, 11b, 12b

Overall, the linear system significantly outperforms the RNNs for the SDOF mass-spring-damper systems. However, for the more complicated MDOF system, both RNNs outperform the linear system, with the pretrained network performing the best by far. Between the RNNs, pretraining either doesn't affect performance or allows the RNN to learn better.

[2]Image taken from `http://www.mathworks.com/help/ident/examples/modeling-an-industrial-robot-arm.html`

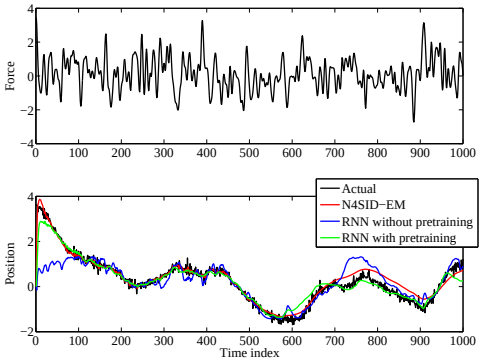
Table 1: Linear mass-spring-damper results

Algorithm	Filtering loss	Inference loss	Total loss
N4SID–EM	**0.0162**	**0.0922**	**0.0562**
RNN without pretraining	0.1629	0.1612	0.1620
RNN with pretraining	*0.0180*	*0.1235*	*0.0735*

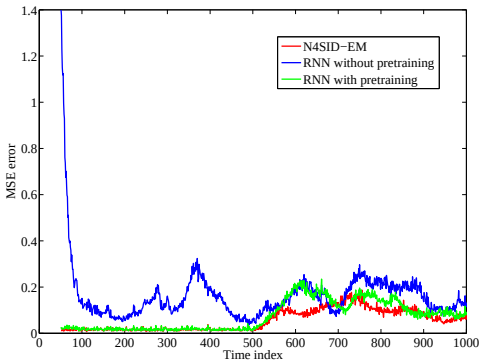
Table 2: Nonlinear mass-spring-damper results

Algorithm	Filtering loss	Inference loss	Total loss
N4SID–EM	**0.0008**	**0.0787**	**0.0418**
RNN without pretraining	0.0252	*0.1745*	*0.1038*
RNN with pretraining	*0.0053*	0.2091	0.1126

5.1 Linear mass-spring-damper



(a) Sample validation set input, output, and predictions)



(b) Validation set loss

Figure 10: Linear mass–spring–damper (filtering at indices 1–500, inference at indices 501–1000)

Here, the pretrained RNN and the learned linear system perform nearly identically, as they overlap a lot in Figures 10a and 10b. However, given its simplicity and ease of generating, the learned linear model is likely preferable.

5.2 Nonlinear mass-spring-damper

The linear system significantly outperforms both RNNs in both filtering and inference. Pretraining does not seem to help the RNN here.

5.3 MDOF mass-spring-damper

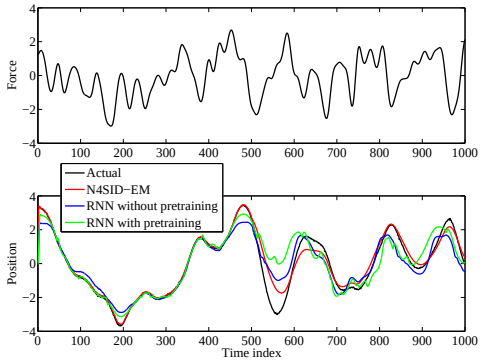
Here, the more complicated dynamics and larger range of inputs and outputs seem more suiting for the RNN, as both RNNs outperform the learned linear model. As well, the pretrained RNN performs ten times better in filtering and 50% better in inference than the other RNN.

6 Conclusion

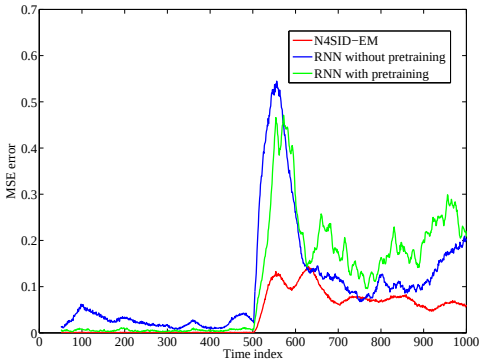
We have presented a framework for training RNNs to perform system identification including a pretraining technique. We have also compared performance between RNNs trained with and without this technique as well as linear systems generated from N4SID-EM. Thus far, pretraining either helps

Table 3: MDOF mass-spring-damper results

Algorithm	Filtering loss	Inference loss	Total loss
N4SID–EM	0.4819	1.0286	0.7580
RNN without pretraining	0.3598	0.8967	0.6310
RNN with pretraining	**0.0367**	**0.5994**	**0.3209**

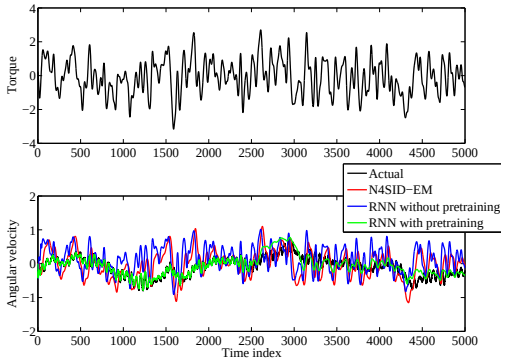


(a) Sample validation set input, output, and predictions)

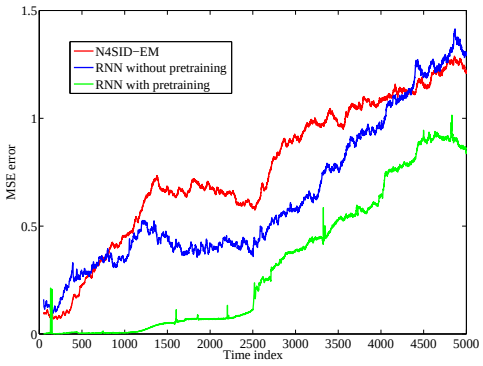


(b) Validation set loss

Figure 11: Nonlinear mass–spring–damper (filtering at indices 1–500, inference at indices 501–1000)



(a) Sample validation set input, output, and predictions)



(b) Validation set loss

Figure 12: MDOF mass–spring–damper (filtering at indices 1–2500, inference at indices 2501–5000)

or does not affect performance of learned RNNs, but for simpler problems, linear systems appear to perform better.

This, however, is far from an exhaustive study on RNNs and system identification. In the future, we'd like to assess how well RNNs perform for more interesting systems (for example, inverted cartpole with partially observed states) that linear systems would likely have trouble with. Also, we would like to try different neural network architectures that might be more adept at learning dynamics. Finally, we would like to use these RNNs in a control environment, like using DDP or MPC with the RNN as a model to invert a pendulum on a cartpole.

References

[1] P. Abbeel, V. Ganapathi, and A. Y. Ng. Learning vehicular dynamics, with application to modeling helicopters. In *Advances in Neural Information Processing Systems*, pages 1–8, 2006.

[2] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1171–1179, 2015.

[3] K. Funahashi and Y. Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks*, 6(6):801–806, 1993.

[4] Z. Ghahramani and G. E. Hinton. Parameter estimation for linear dynamical systems. Technical report, Technical Report CRG-TR-96-2, University of Totronto, Dept. of Computer Science, 1996.

[5] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[6] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. volume 31, pages 855–868. IEEE, 2008.

[7] A. Graves, A.-R. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[8] J. Ko and D. Fox. GP-BayesFilters: Bayesian filtering using Gaussian process prediction and observation models. *Autonomous Robots*, 27(1):75–90, 2009.

[9] I. Lenz, R. A. Knepper, and A. Saxena. DeepMPC: Learning deep latent features for model predictive control. In *Robotics: Science and Systems*. Rome, Italy, 2015.

[10] L. Ljung. *System Identification: Theory for the User*. Prentice Hall, 1987.

[11] A. Punjani and P. Abbeel. Deep learning helicopter dynamics models. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3223–3230. IEEE, 2015.

[12] T. Tieleman and G. Hinton. Lecture 6.5-RMSprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[13] P. Van Overschee and B. De Moor. *Subspace identification for linear systems*. Springer Science & Business Media, 2012.