Exploration of Long Short-Term Memory Networks for Time Series Forecasting

Eliot Liucci

Department of Mathematical Sciences
Montana State University

Date of completion here as Month Day, Year

A writing project submitted in partial fulfillment
of the requirements for the degree

Master of Science in Statistics

APPROVAL

of a writing project submitted by

Eliot Liucci

This writing project has been read by the writing project advisor and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the Statistics Faculty.

Date	Dr. John Smith Writing Project Advisor
Date	Dr. Katharine Banner Writing Projects Coordinator

Abstract

Long Short-Term Memory (LSTM) networks are a special case of Recurrent Neural Networks (RNN) that allow for past information to make informed predictions. LSTMs are able to do this without the exploding/vanishing gradient problem that can occur with RNNs. The data used were collected within the Everglades National Park as water depth time series. In this paper, a comparison of LSTMs is made to other methods such as ARIMA models and Holt-Winters models.

Contents

1 Introduction

Artificial intelligence has become the buzz word of modern computing. While the inner workings of most machine learning techniques are overlooked by many disciplines, the end results speak for themselves. Machine learning techniques allow for adaptable models to be applied to a variety of problems. Traditional statistical methods require a deep understanding of the mathematics behind the algorithm, which can lead to an increase in the time to obtain a result. Machine learning algorithms do not require as deep of an understanding, but the reduced level of understanding required comes at the cost of interpretability. Parameters within models like Least Squares Regression can be interpreted directly, allowing for researchers to obtain a deeper understanding of what the model achieves.

Deep learning is a term used to describe the use of neural networks, which will be the focus of this paper. These particular models involve an input layer, a hidden state comprised of 1 or many layers, and an output layer. A visualization of a neural network is shown in Figure 1.

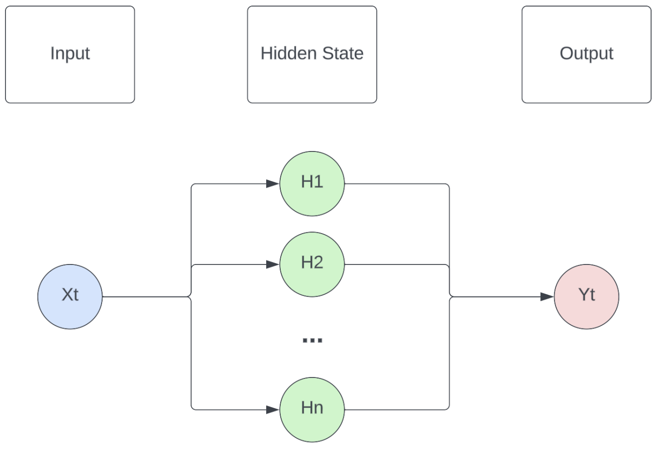


Figure 1: A flowchart depicting a single-layer Neural Network. The input layer (blue) contains a single node, indicating a single value would be taken as an input. The hidden state (green) contains a single layer of n nodes. The output layer (red) contains a single node, indicating a single value would be received as an output. Graphic made with LucidChart.

The nodes within a neural network, specifically within the hidden layer, take an input value from the input layer and apply an activation function, which typically convert any given input to a value within a given range based on the activation function. A visualization is shown for the Linear, Rectified Linear Unit (ReLU), Sigmoid, and Hyperbolic Tangent (Tanh) activation functions in Figure 2. This process is important as it allows for the outputs of each node to be normalized which can help with the training process for the weights and biases in the network. For example, if inputs were recorded as meters, a 1,000 meter input would have a larger impact on the output than if the units were recorded as kilometers.

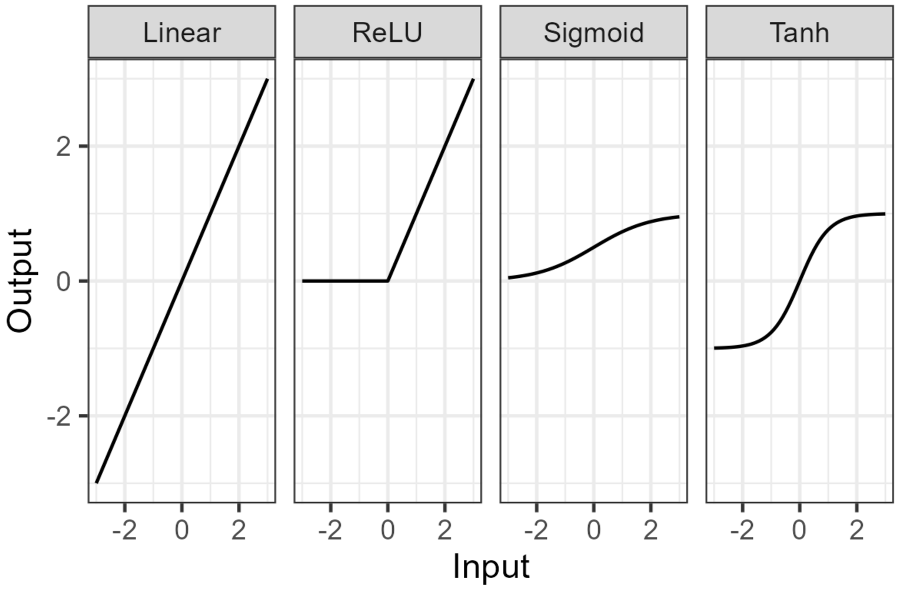


Figure 2: A comparison of common activation functions. Note how the range of the inputs compares to the range of the outputs for different activations.

In addition to activation functions, each connection between nodes has a weight and bias. For example, the connection between H_1 and X_t in Figure 1 would take the form:

$$H_1 = \sigma[w_1 * X_t + b_1]$$

where σ indicates the activation function being applied, w_1 indicates the weight, and b_1 indicates the bias. The weights and biases in a network are what get tuned in the training process.

Training of neural networks occurs through the process of backpropogation, which is an application of the chain rule in calculus to determine how the output changes based on the change of a single parameter, or trainable value (weights and biases). In practice, this means that the change in the prediction can be written as a function of the change in each parameter. The prediction of y is compared to the observed value of y in order to approximate a gradient, where each dimension represents a parameter. Then, gradient descent is used to find a minimum error, or loss, in the predictions.

1.1 Recurrent Neural Networks

Recurrent neural networks, or RNNs for short, are essentially multiple neural networks that feed into each other a specified number of times (Rumelhart et al., 1986). Similar to traditional neural networks, these networks have an input layer, a hidden state, and an output layer. As recurrent patterns can be difficult to visualize, these networks are often visualized in an 'unrolled' state. This is visualized in Figure 3, where instead of the network looping back into itself 3 times, the 3 different time states are visualized as separate networks.

Recurrent neural networks are very useful when working with time series as observations depend on previous observations. A key issue that can arise when using too many time points to predict a single output is an exploding/vanishing gradient, which occurs when a weight associated with time point 1 ends up being multiplied by itself n times to predict n time points ahead. Using Figure 3 as an example network with a single layer and a single node in each hidden state, the

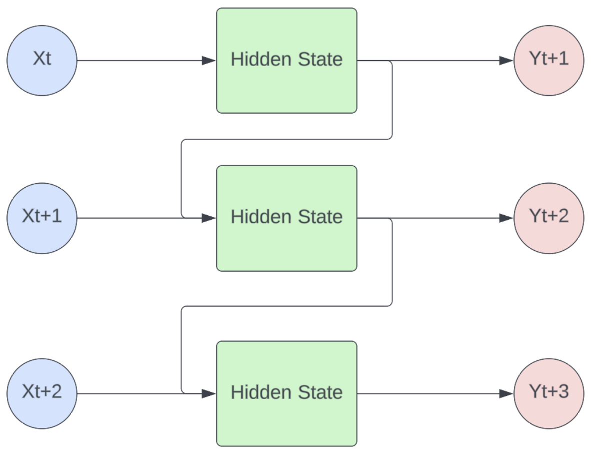


Figure 3: A visualization of an 'unrolled' recurrent neural network. This specific example uses 3 time-points of information to get a prediction.

prediction for y_1 would look like:

$$y_1 = \sigma[w_1 \cdot x_1 + b_1] \cdot w_2 + b_2$$

Then, using that prediction in the following time points prediction:

$$y_2 = \sigma[w_1 \cdot x_2 + w_3 \cdot \sigma[w_1 \cdot x_1 + b_1] + b_1] \cdot w_2 + b_2$$

Similarly, for the final prediction:

$$y_3 = \sigma[w_1 \cdot x_3 + w_3 \cdot \sigma[w_1 \cdot x_2 + w_3 \cdot \sigma[w_1 \cdot x_1 + b_1] + b_1] + b_1] \cdot w_2 + b_2$$

In the equations above, w_1 represents the weight applied to the input, w_2 represents the weight applied to the output of the hidden state, and w_3 represents the weight applied to past predictions in the current prediction. Similarly, b_1 represents the bias being added to the input and b_2 represents the bias added to the hidden layer's output. Note that w_1 is used in each equation for the weight on the input as we have assumed a single layer and single node network. The issue with gradient approximation appears if w_3 is a value above 1, which causes the value of x_1 to have a much larger impact on the final prediction than other inputs. To solve this, 'memory' can be added to recurrent neural networks.

1.2 Long Short-Term Memory Networks

Long short-term memory networks are very similar to recurrent neural networks, but they address the issue of gradient approximation by introducing memory cells and a cell state (Hochreiter and Schmidhuber, 1997). A visualization of these networks is shown in Figure 4. The cell state acts as the 'long term' memory, making use of all previous information. The only increase that can occur within the cell state is additive, which limits the impact of long-term dependencies that caused issues within RNNs. The memory cells act as the 'short term' memory, making use of mostly the current input and previous output.

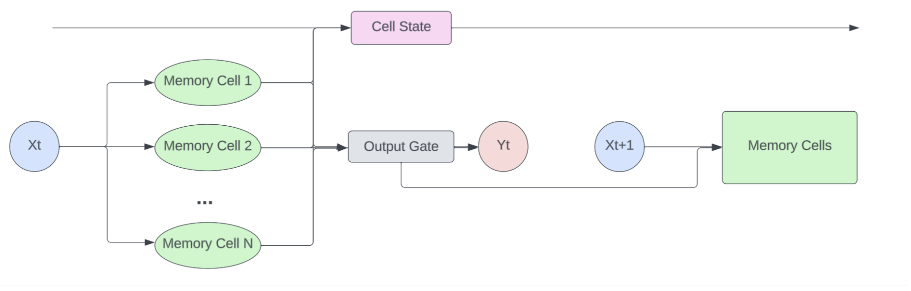


Figure 4: A visualization of long short-term memory networks.

The memory cells are more involved than the typical hidden states of RNNs (see Figure 5). This is because of the way each prediction is used for both the long-term and short-term memory. The current input x_t and previous output y_{t-1} are used in more ways than simply getting the next prediction. The forget gate determines what percentage of the previous cell state is remembered. By using a sigmoid activation function, the forget gate value will be between 0 and 1. Gates function as filters, determining what percentage of the input should actually be kept

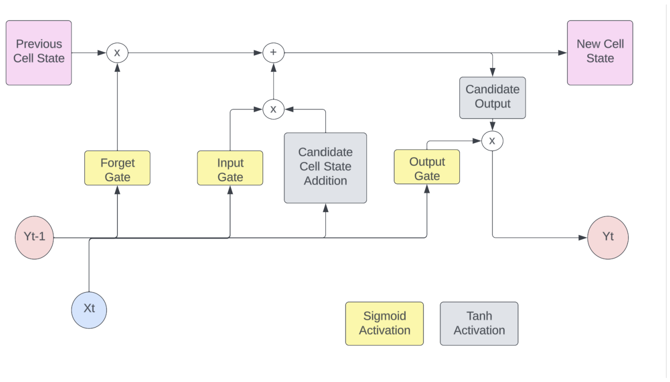


Figure 5: A visualization of the memory cells in LSTMs.

in the output. Sigmoid activations work well here as they map any values on the real number line to a value between 0 and 1. The cell state is then modified based on the candidate cell state addition, which is then multiplied by the input gate value. This gate acts similarly to the forget gate in the sense that it determines what percentage of the candidate cell state addition is actually added to the cell state. Finally, the cell state is fed into a hyperbolic tangent activation and multiplied by the output gate to determine that final output value y_t. The purpose of using the hyperbolic tangent activation is that it allows the output to be positive or negative based on if it is an increase or a decrease, respectively.

Mathematically, the memory cell can be written in notation used previously (adapted from colah (2015)). The forget gate f_t makes use of a sigmoid activation and has its own corresponding weights and bias, denoted W_f and b_f, respectively.

$$f_t = \sigma[W_f(y_{t-1}, x_t) + b_f]$$

The input gate i_t functions similarly, with the notation of a subscript i.

$$i_t = \sigma[W_i(y_{t-1}, x_t) + b_i]$$

The new candidate values $\tilde{C}_t$ to be added to the cell state use a hyperbolic tangent activation with the subscript c on the weights and bias.

$$\tilde{C}_t = tanh[W_c(y_{t-1}, x_t) + b_c]$$

The updated cell state C_t is the product of the forget gate f_t and the previous cell state C_{t-1} plus the new candidate cell state values C_t multiplied by the input gate i_t.

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

Similar to previously mentioned gates, the output gate o_t uses a sigmoid activation with the subscript o on the weights and bias.

$$o_t = \sigma[W_o(y_{t-1}, x_t) + b_o]$$

The output value y_t is calculated as a proportion (determined by the output gate) of the new cell state.

$$y_t = o_t \cdot tanh[C_t]$$

1.3 EVER Data

In order to train a complex network like LSTMs, a large data set is required. Ideally, the data would have low missingness, as missing values can impact training effectiveness (this is true for all networks, although the issue becomes more pronounced when data are time-dependent). Additionally, the data should not have any seasonal increase over time. Due to these requirements, the training data were selected to be water depth time series collected within the Everglades National Park (EVER). There are 52 monitoring stations present, all with various levels of missingness. The chosen station, P33, had less than 1% missingness and had only a minor increasing trend (see Figure 6).

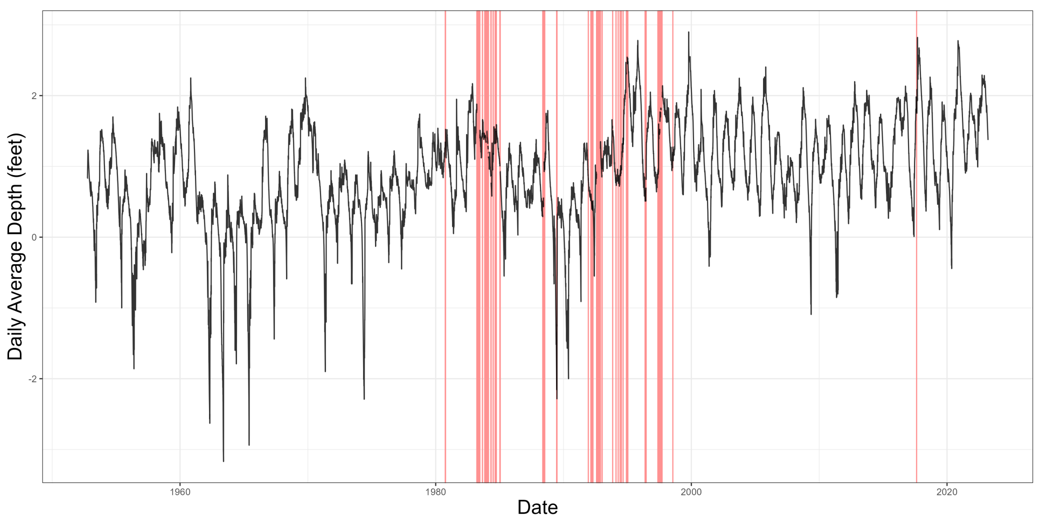


Figure 6: A line plot of the time series used to train the LSTM in this project. Red vertical bars indicate a date with a missing value.

Due to the present although low missingness, an interpolation method had to be chosen to impute those missing values. Several candidate algorithms were tested, but eventually the decision to use a Kalman filter was made (Kalman, 1960). The line plot in Figure 7 displays the imputed values for the date range of 1960 to 2000.

2 Methods

Before discussing the specifics of the model selection process, some terminology should be discussed. The *batch* used in training is the number of inputs that are used for a single prediction. For example, with a batch size of 31, the network is using the past month of information to predict the 32^{nd} day's information. When discussing an *LSTM layer with n nodes*, this is in reference to n of the nodes depicted in Figure 5, where each node takes information from the entire batch. *Activation functions* were discussed previously in Section 1, but the main idea is that they are functions that restrain the range of values, mapping each input to a corresponding output. A *dropout layer* is not so much a layer in the physical sense, but is rather a method of regularization applied to the previous layer. For example,

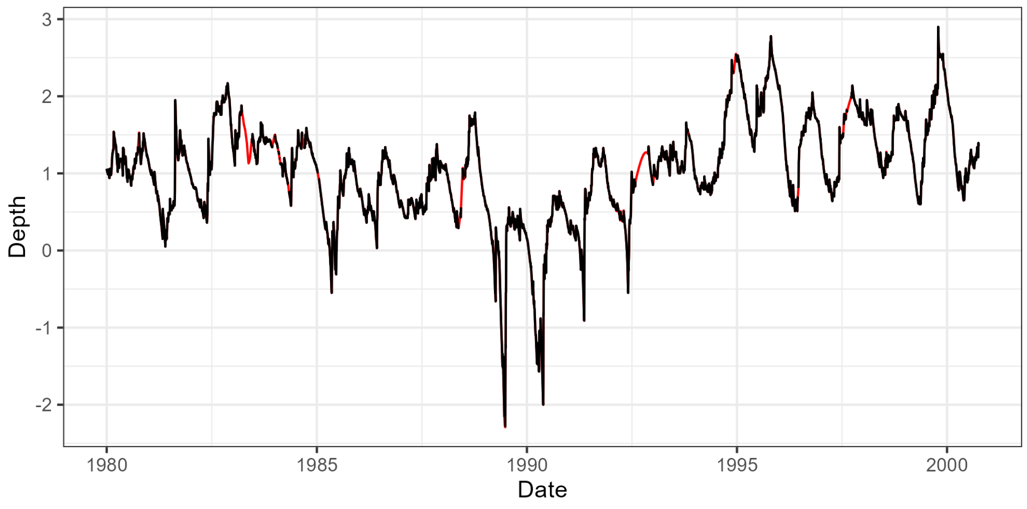


Figure 7: A line plot of the time series used to train the LSTM in this project. Red lines indicate the imputed values using the Kalman filter. Note that this plot focuses on 1980 to 2000, despite imputation being performed across the entire time series.

a dropout layer with a rate of 5% means that in each training iteration, there is a 5% chance of any node being turned off. This process helps avoid overfitting, which is a common issue with deep learning models.

To create the LSTM network, the *Keras* framework within Python was used (Chollet et al., 2015). There are several powerful tools for building machine learning models, but *Keras* was chosen because of the ease of use and ability to take advantage of hyperparameter tuning when optimizing the model. The initial steps of model creation involved imputation (discussed in Section 1.3), creating a simple model that could be made more complex over time, and splitting the data into a testing and training set. The training set was chosen to be the first 90% of the data between 1995 and 2023, leaving the remaining 10% for testing. The initial model contained a single LSTM layer with 128 nodes, each of which used the ReLU activation function for the 'Candidate Output' and 'Candidate Cell State Addition' in Figure 5. Note that the gates used the sigmoid activation function, which is the default in *Keras*. Once the initial model was trained, code was adapted from Hebbar (2021) to get out-of-sample predictions. All code is available on GitHub.

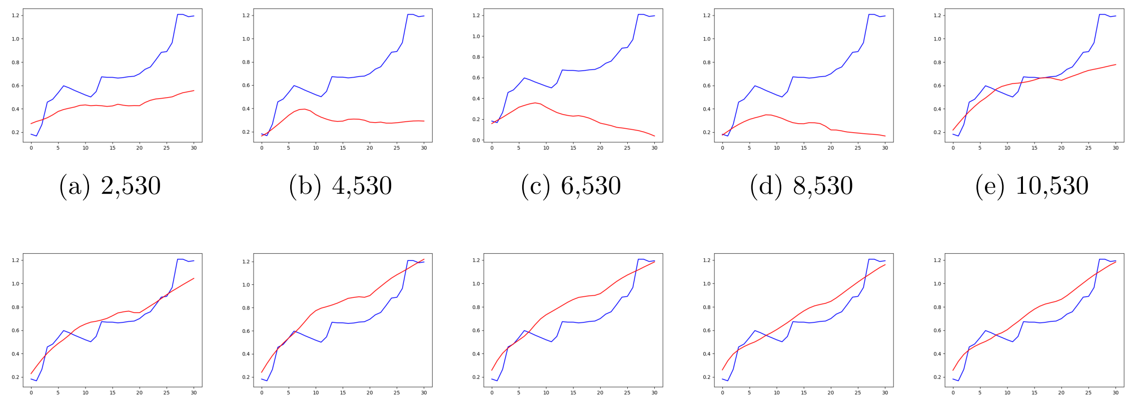
Once a base framework and method for validation was created, the structure of the model could be tweaked. This was done by implementing a random search across a set parameter space. The base model contained 3 LSTM layers and 3 Dropout layers. Parameters that were defined to be searched within each LSTM layer were the number of nodes (minimum of 8 with a maximum of 64) and the activation function (either ReLU or Tanh). For the Dropout layers, the dropout rate was chosen to be either 1%, 5%, 10%, or 15%. Fixed parameters include the kernel regularizer (L1 norm with a rate of 0.001) and the activity regularizer (L2 norm with a rate of 0.001).

The random search comprised of 50 randomly chosen combinations that were trained for 30 iterations each. The best model was chosen as the model with the lowest mean squared error when compared to the testing data set. The best model had the following structure;

- LSTM Layer: 17 Nodes, ReLU Activation
- Dropout Layer: Rate of 15%
- LSTM Layer: 11 Nodes, Tanh Activation
- Dropout Layer: Rate of 15%
- LSTM Layer: 37 Nodes, Tanh Activation
- Dropout Layer: Rate of 5%

This model was trained for an additional 20,500 iterations. A figure containing the training progress is shown in Figure 8. A key idea to note from the training process is that after about 12,000 iterations, the model performance becomes fairly constant with only minor improvements over time.

In order to get a sense of the variability that can arise in the training process, 5 models were initialized with the same structure (determined by the random search



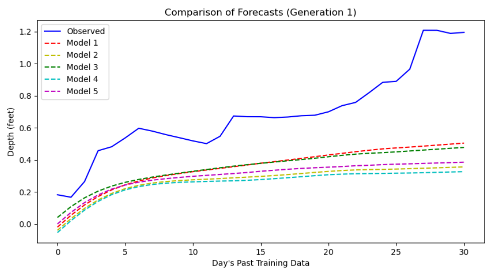
(a) 2,530 (b) 4,530 (c) 6,530 (d) 8,530 (e) 10,530

(f) 12,530 (g) 14,530 (h) 16,530 (i) 18,530 (j) 20,5530

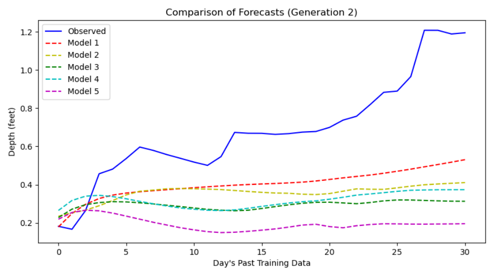
Figure 8: A series of plots obtained from the training process. Each plot shows the predicted water depths 31 days past the training set. The red line indicates the predictions from the LSTM and the blue line indicates the observed water depth. The number below each plot indicates how many generations the LSTM was trained for when the predictions were made.

of hyperparameters). These 5 models were all trained on the same training data and evaluated on the same testing data. Comparisons of forecasts to the observed time series are shown in Figure 9. Looking at plot a), it can be seen that the 5 models perform similarly after 2,000 iterations of training. After 4,000 iterations, the 5 models start to break up as they all find different weights and biases in the training process. After 6,000 iterations, model 2 outperforms all other models in the first 10 days of forecasting. After 8,000 iterations, model 2 continues to improve while models 3 and 5 gain considerable improvement. At 10,000 iterations, models 2, 3, and 5 are by far the best performing models in the 31 day forecast while models 1 and 4 struggle to correctly predict after 3 days.

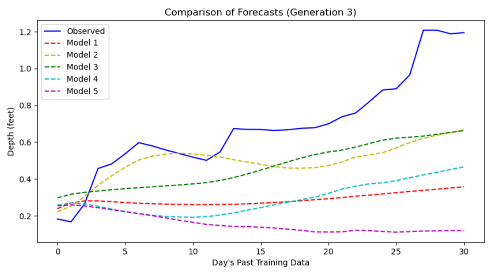
This graphic is not a perfect measure of variability in predictions as each model contains different weights and biases despite having the same structure. The purpose of these graphics is to show how the LSTMs can vary when given the same training data, same structure, and same training time. If validation data were not available, it would be difficult to trust the model's predictions as some perform great while others fall short.



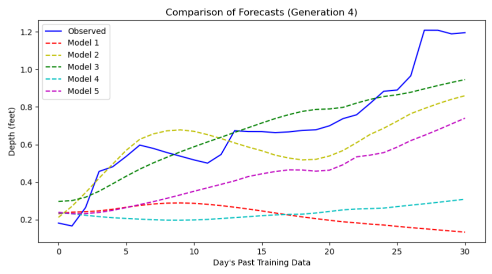
(a) 2,000



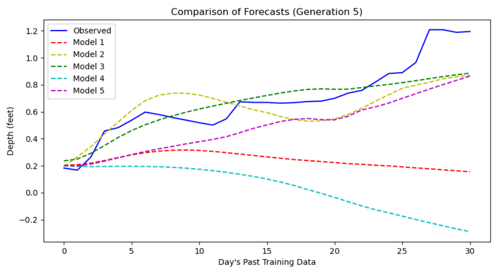
(b) 4,000



(c) 6,000



(d) 8,000



(e) 10,000

Figure 9: A line plot comparing the 1 month out-of-sample predictions for 5 models initialized with the same framework. Each model was trained on the same training data for the number of iterations displayed below the plot.

3 Results

While the overall performance of the LSTM is impressive, it is difficult to make statements about just how impressive without comparing the predictions to other popular forecasting methods. As alluded to in Section 1, an Autoregressive Integrated Moving Average (ARIMA) and a Holt-Winters Exponential Smoothing model were fit on the same training data used for the LSTM. The Holt-Winters model used triple exponential additive smoothing and the ARIMA model was fit using the 'auto_arima()' function within the 'tsa' package, resulting in a 4^{th} order auto-regressive model. Each model was used to forecast 31 days past the training data and were compared to the observed water depths (see Figure 10).

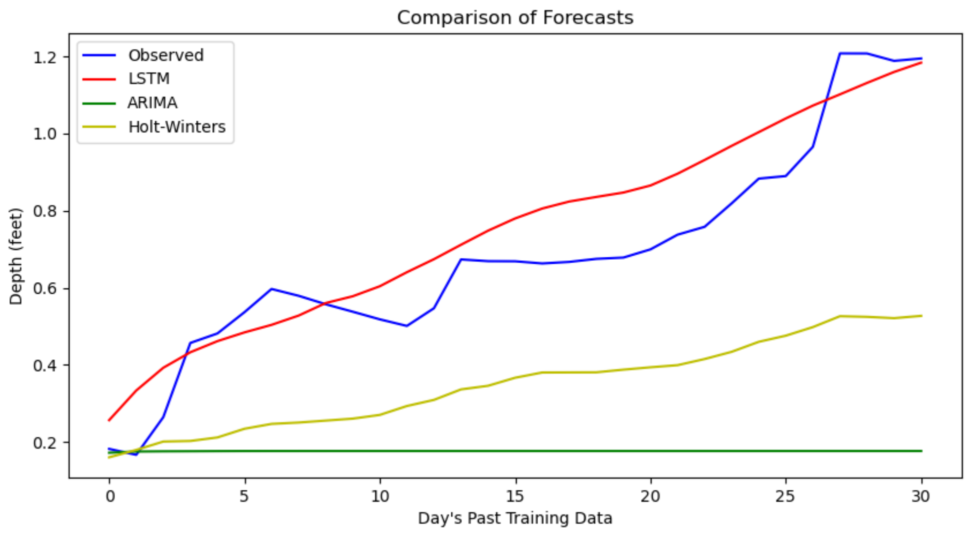


Figure 10: A comparison of 31 day forecasts between the 3 models. Each model was trained on the same data.

While it may seem strange that the ARIMA model predicts a constant trend, this can be explained by the estimated coefficients for the autoregressive components in the model. This constant prediction indicates that the water depth is likely to come back down after a while.

3.1 Reflection

While it seems as though the LSTM outperforms both models, the decision cannot be made without some level of uncertainty on the forecasts obtained. Prediction intervals can be obtained with ease for the Holt-Winters and ARIMA model, but such intervals are not as easily obtained for the LSTM. To do this, the framework of the LSTM would have to be retrained on bootstrapped samples of the original training data and used to make predictions. This could be repeated 1,000 times to obtain prediction intervals. This process, however, would require a lot of time and is out of the scope of this paper. As the LSTM being compared required over 20,000 iterations for a single prediction, the process of obtaining uncertainty for the predictions would take quite some time.

Another issue that arises is the loss of interpretation of these results. For example, the ARIMA model had a 4^{th} order auto-regressive component, which indicates that the water depth at any given day is mostly related to the water depth of the past 4 days. Inference like this cannot be made with the LSTM.

For future research, uncertainty intervals should be obtained to determine if the LSTM is able to obtain higher levels of certainty regarding where a time series will be in the future when compared to other methods. Another step that could be taken for future work is to compare additional methods for time series forecasting. While ARIMA and Holt-Winters are common and effective, there is no shortage of methods that can be used to forecast time series.

Another interesting route that could be explored is to implement a genetic algorithm to randomly mutate the a subset of high-performing models to create new generations. In Figure 9, it can be seen that 3 of the 5 models perform very well. A genetic algorithm would allow for some of the components of each model to be taken and turned into a new generation of models. From there, the next top performers could be selected and used for future generations. This process would work best when all models have differing structures, although that was not performed for this analysis.

References

Chollet, F. et al. (2015). Keras. `https://keras.io`.

colah (2015). Understanding lstm networks.

Hebbar, N. (2021). Time series forecasting with rnn(lstm)— complete python tutorial—.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation, parallel distributed processing, explorations in the microstructure of cognition, ed. de rumelhart and j. mcclelland. vol. 1. 1986. *Biometrika*, 71:599–607.