Predicting Weather Using Recurrent Neural Networks

Cory Harris
Erik Jonsson School of Engineering and Computer Science
University of Texas at Dallas
Richardson, USA
clh200002@utdallas.edu

Mathew Singsing
Erik Jonsson School of Engineering and Computer Science
University of Texas at Dallas
Richardson, USA
mes180005@utdallas.edu

Ryan Mendoza
Erik Jonsson School of Engineering and Computer Science
University of Texas at Dallas
Richardson, USA
rpm200001@utdallas.edu

Rygiel Corpuz
Erik Jonsson School of Engineering and Computer Science
University of Texas at Dallas
Richardson, USA
rsc190002@utdallas.edu

***Abstract*— This report documents our creation of a recurrent neural network using long short-term memory cells with the goal of accurately predicting the temperature, humidity, and pressure for the following day. Using a total of 43,487 entries recorded across the last ten years, (1993-2023) our dataset contains data on daily temperature, precipitation, humidity, and pressure in Dallas, Texas. This project was done in Python, with all primary methods and algorithms coded without the use of libraries. All libraries utilized were used solely for the purpose of pre-processing the dataset and analyzing the final data.**

Keywords—weather, prediction, recurrent neural network, python,

I. Introduction

Forecasting weather has been of great importance throughout history. The implications of accurate meteorological conditions on any given day, affects the success of supply chain logistics, military operations, and even daily commute choices. With the rise of more advanced learning methods in Machine learning, the landscape of meteorological predictions has transformed. Initially the focus was on heavy computational methods like the advanced Numerical Weather Prediction (NWP) models [1]. These methods required expensive supercomputing due to the use of nonlinear differential equations to simulate atmospheric conditions. But with the chaotic nature of weather as well as the cost prohibitive computing power necessary, applying Machine learning methods is a growing paradigm to predict meteorological conditions. Specifically Recurrent Neural Networks (RNN) have proven to do well at modeling time series data such as weather [2].

This paper aims to implement a Recurrent Neural Network from scratch without using prepackaged libraries for the Recurrent Neural Network itself. This paper will detail the implementation of the manually coded RNN algorithm. This algorithm will be used to predict temperature, precipitation, humidity, and pressure of the atmosphere in data taken from Dallas Texas over the course of a six-year period between 2016 to 2023.

II. Study of Recurrent Neural Networks

Recurrent neural networks (RNNs) are a type of deep learning algorithm best suited for sequential data processing. Like other neural networks, RNNs are comprised of a series of perceptrons, or neurons, grouped together to form layers. Information is passed through an input layer, through a series of hidden layers, and finally through an output layer in a feed-forward direction. In a standard neural network, all inputs and outputs are typically independent of one another. Unfortunately, depending on the application, it is often necessary for one neuron to know, or remember, the information from a previous one [3].

RNNs are specifically designed to implement this feature. Unlike other traditional feed-forward neural networks, layers have connections that form directed cycles. In other words, the output of each processing node is saved and then fed back into the model. This allows them to maintain memory of previous inputs and makes them effective in a variety of applications, such as natural language processing, time series analysis, and speech recognition. Given its prowess with working with sequential data, it will serve particularly useful for our weather prediction report [2].

The RNN algorithm is governed by the forward pass and the backward pass (backpropagation) through each layer in the neural network. Both processes are responsible for the update of the model's parameters to improve the ability to make accurate predictions on unseen data.

In the forward pass, hidden states and output are computed at each time step. The hidden state, denoted as $\boldsymbol{h_t}$, can be computed as:

- $\boldsymbol{h_t = f(W_{hx}x_t + W_{hh}h_{t-1} + b_h)}$

Where:

- $\boldsymbol{W_{hx}}$ is the weight matrix for the input
- $\boldsymbol{x_t}$ is the input
- $\boldsymbol{W_{hh}}$ is the weight matrix for the recurrent connection

- $\boldsymbol{h_{t-1}}$ is the previous hidden state
- $\boldsymbol{b_h}$ is the bias
- $\boldsymbol{f}$ is the activation function

The output at time $\boldsymbol{t}$, denoted as $\boldsymbol{y_t}$, can be computed as:

- $\boldsymbol{y_t = g(W_{yh}h_t + b_y}$

Where:

- $\boldsymbol{W_{yh}}$ is the weight matrix for generating output
- $\boldsymbol{h_t}$ is the hidden state
- $\boldsymbol{b_y}$ is the bias
- $\boldsymbol{g}$ is the activation function

In the backpropagation step, the gradient of the loss with respect to each parameter is calculated using backpropagation through time (BPTT). The product of the learning rate and these computed gradients are then subtracted from the current weight, with the goal of reducing loss and improving overall performance.

One of the main issues with RNNs is their susceptibility to vanishing and exploding gradient problems. During back propagation through time, the gradients may tend to become very large, known as exploding gradients. Likewise, vanishing gradients occur when gradients become extremely small, which may result in the model taking an excessive amount of time or to stop learning entirely. One way to combat this is through the use of long short term memory (LSTM) cells. In short, utilizing this mechanism allows for the control of information and gradients, allowing for better learning over longer sequences of data.

III. Methodology

A. Dataset description

The dataset we chose to use was downloaded from a third-party website called visualcrossing.com/weather/weather-data-services. We hosted the downloaded data on a public GitHub repository to be imported into the Google Colab that we are using for the RNN algorithm implementation. The data was collected by individual weather stations located around the globe. The Visual Crossing System aggregates this data and uses interpolation to calculate the weather for nearby non-reported areas. The weather stations themselves were official government meteorological organizations such as NOAA in the United States, DWD in Germany, and JMA in Japan. For this paper, we only pulled data for Dallas, Texas and, therefore, the primary data collection was done by the NOAA. The data was collected in real time at fixed intervals. We adjusted the data to be at daily intervals over the course of 1993 until 2023.

B. Data preprocessing

The process for data preprocessing was multiple steps. This stepwise process was crucial to ensuring clean data for our RNN algorithm to digest. To begin, the data needed to be forward filled in to replace any missing values and to ensure continuity. Pandas served to streamline this process with various built-in methods. The datetime column of the data set was converted into a pandas datetime object allowing time-series analysis. The relevant features to select were temperature (labeled as 'temp'), humidity, precipitation (labeled as 'precip'), and sea level pressure (labeled as 'sealevelpressure'). The variable used as a prediction target was the temperature of the next day labeled as 'tomorrow_temp'.

Once the initial data preprocessing was complete, the data was normalized using a standardScaler function included in the sklearn.preprocessing library. The goal was to ensure the mean was zero and the standard deviation was one, which helped for the convergence speed during the neural network training.

C. Structure of the RNN

The type of algorithm we implemented was that of a vanilla RNN. The algorithm consisted of a forward pass through the input, hidden, and output layers. As well as a back propagation through time (BPTT) in the reverse order. The best result we found was utilizing a single input layer that contained four neurons, one for each of the four inputs, then a single hidden layer that contained four neurons. Lastly, an output layer that had one neuron for which the prediction would be returned. As for the configuration within the RNN, we utilized the *tanh* activation function for both our forward pass and BPTT. This helped alleviate any gradient instability and ensured non-linearity in transformation.

D. Model implementation

We programed our RNN utilizing python within Google's Colaboratory, which is a product of their research team. This allows anyone to collaborate with each other by editing, compiling, and executing python code within the browser. We implemented our RNN using several functions to handle the various components of training and evaluating the model. Key ones included our forward pass and BPTT functions, which do the bulk of the work of building and training the model.

The purpose of 'forward_pass' was to propagate the input through the RNN and compute the activation of each neuron of the hidden layer. Ultimately the return of this function was the overall output of the matrix multiplications above while maintaining state transitions within hidden layers. The core function is to read in the 'input_data' sequence to process as well as 'rnnLayers' which contains all our weights and biases for each layer. We then iterate through every data point in our sequence to calculate the hidden state using our *tanh* activation function and compute the output for that layer to then be passed to the next. It repeats this process until we reach the final output layer, which is returned.

Another custom function necessary for implementation was the 'BPTT' used for Back Propagation through time. The function changed the parameters in reverse chronological order. It ensures that the previous time steps influence is accounted for during the learning process. Some key points include the use of the *tanh* activation function once again to update the hidden layer gradients. With this in mind, the

prediction error could be reduced by adjusting weights for optimization of the network matrices.

The 'train_epoch' was written to help assess performance and change parameters during the training process over many iterations. The goal was to assess and reassess based on the error given by the 'BPTT' function. Additionally, the 'calculate_MSE' function would calculate the Mean Squared Error (MSE) on the data set that is passed to it. Its goal is to assess and quantify the accuracy of the model predictions.

IV. Results and Analysis

A. Training the RNN

The process of training our Recurrent Neural Network involved several steps. For the input features that are used for training, we had the daily temperature, humidity, precipitation, and sea level pressure. The output we are predicting is the next day's temperature. To begin training, we split the data into training, validate and test sets. For our split we used 80 percent train, 10 percent validate, and 10 percent test. A visualization of this is shown in Figure 1. Next, we

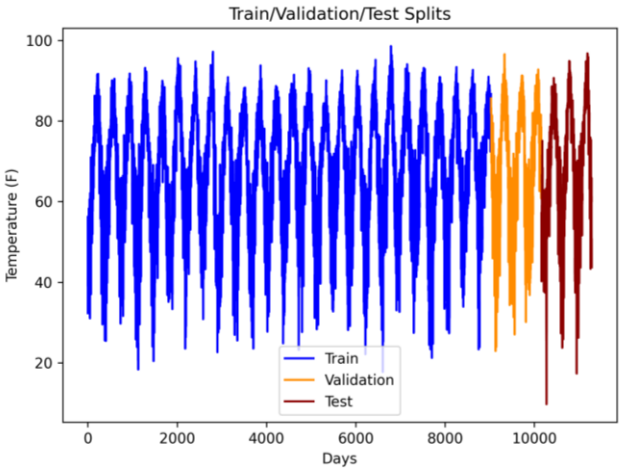

Fig 1. 80, 10, 10 split between train, validation, and test data

separate the data into sequences to be passed through the forward pass. Following this, we perform our back propagation through time to update the weights and repeat for the specified number of epochs. Once we reach the epoch limit, or our Mean Squared Error triggers the tolerance, we stop the training and have our completed model that is now ready to use.

B. Evaluation and Validation

For the evaluation of our RNN, our primary metric used was Mean Squared Error. We chose it because it is a standard measure of regression models. This is because the average squared difference between predicted and actual values gives a good indicator of accuracy. It is especially suited for continuous data sets. By using MSE we can assess the model in terms of its accuracy in predicting the output. The use of MSE during both the training and validation phase of the RNN ensures consistency in the way we keep track of the model's performance. By comparing the MSE across the training and validation splits, we can get a good idea of how well the model is performing in terms of overfitting and if the model is able to accurately generalize to the validation and test data. This also allows us to edit the model if we find it to be memorizing the training data and its noise instead of recognizing its patterns.

C. Results

Table 1 showcases the mean squared error (MSE) through various number of iterations (epochs) with one input, hidden, and output layer for a total of three layers. We decided on a neuron count of four for the input and hidden layers and a single neuron for the output layer. Additionally, we settled on a learning rate of 0.00001. Out of multiple runs, we found that these were the optimal hyperparameters for our model.

As shown in the table, the MSE steadily decreases as the number of epochs increases, stabilizing at around 63 epochs. This is a good indication that our model is learning and gradually improving its ability to make accurate predictions. Additionally, the validation MSE aligns closely to our training MSE, a good sign that we have avoided overfitting and that our model generalizes well to unseen data.

TABLE I. Mean Squared Error

Epochs	**MSE**	
	Training MSE	***Validation MSE***
0	3528.660	2417.526
5	302.859	294.730
10	91.915	93.494
15	59.187	62.692
20	47.288	51.586
25	42.080	46.622
30	38.844	42.404
35	36.071	40.626
40	33.468	38.102
45	31.452	36.330
50	30.333	35.374
55	29.887	34.942
60	29.791	34.774
63	29.854	34.774

Examining the Mean Squared Error loss curve of our training model, as shown in Fig. 2, revealed better insight into how our model was performing when it is iterating and improving over the course of time. It shows a clear visualization that the forward and backwards propagation are working as intended and it is improving the weighting after every epoch to slowly normalize the curve to a low error. This was the first indication that our model was working correctly as it shows a continuous trend of improvement as our MSE of every epoch gets lower and approaches a minimum.

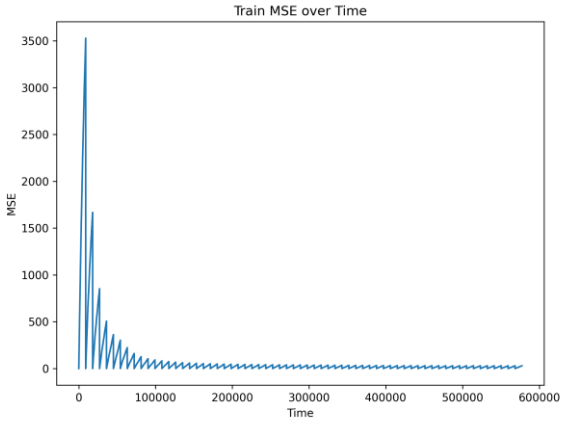

Fig 2. Mean Squared Error of model training.

Figure 3 plots the predicted and actual values of the training data. The graph indicates the daily temperature over the course of around 9000 days. As indicated by the graph, the predicted values of the model are close to the actual values. This is a positive sign and, combined with an MSE of 29.854, indicates that our model has learned the appropriate patterns and relationships present in the training data.

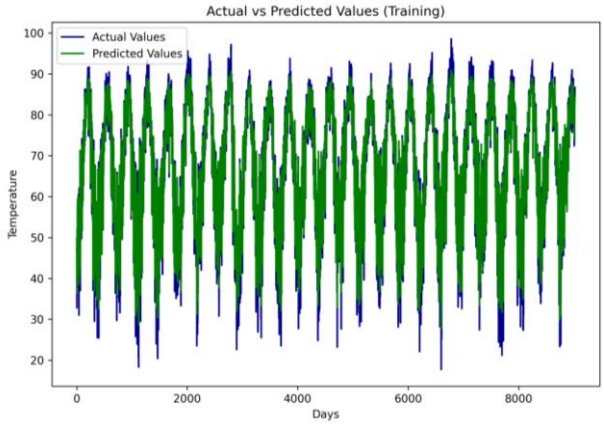

Fig 3. Actual versus predicted values of training dataset.

The next in our analysis is to graph the values for the validation dataset and evaluate the plot. Taking a look at Figure 4 we can see a very similar story to the training set with our validation dataset. A very similar plot combined with a validation MSE of 34.774 tell us that the model is well trained.

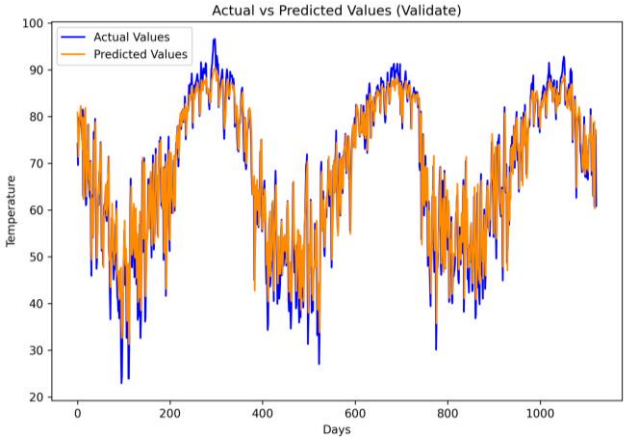

Fig 4. Actual versus predicted values of validation dataset.

Now that we have a low training and validation MSE we are able to move on to the final analysis and of the testing data. The graph of this analysis can be seen in Figure 5. The key

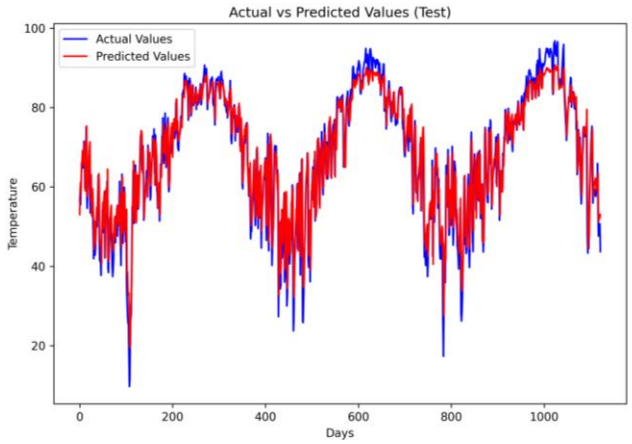

Fig 4. Actual versus predicted values of validation dataset.

takes away from the evaluation of the testing data set is to test our model against data that it has never seen before. This is an important step in ensuring that the model isn't overfitted and that it has achieved its intended purpose.

Taking a look at the graph we can see a great amount of overlap, combined with a test MSE of 31.628 we can conclude that the model works as expected and performs well above expectations. Since MSE is a squared value, this means that taking the square root of it tells us about the actual variance in degrees Fahrenheit, which in this case is around 5.623 degrees Fahrenheit for our test dataset. This is very good for the amount of data we have combined with the unpredictability of weather conditions.

V. Future Work and Conclusion

A. Future Work

Moving forward, we would like to explore ways we can further improve and expand on the weather temperature prediction model. One aspect to consider is to incorporate additional features to the model to enhance its prediction capabilities and potential. Weather is affected by a lot more factors than what was used in the model. For example, we can consider other parameters such as wind speed. A more adaptive model will lead to improved forecasting prediction performance.

Furthermore, we may implement more advanced recurrent neural network architectures such as Long Short-Term Memory. This would lead to better long-term modeling especially for cases like time series predictions. This can significantly help reduce any vanishing gradient problems.

B. Conclusion

Ultimately, after testing various hyperparameters, we were able to get our model down to an MSE of 29.854. This illustrates the ability for the model to predict the actual temperature per day visualized by the figures mentioned previously. This study provided us with an insight on

temporal patterns in the data, especially for specific months and seasons. Through this project, we were able to showcase the implementation and evaluation of a recurrent neural network for a time series forecast, offering a way to predict the temperature of the day.

REFERENCES

[1] "Deep Learning-Based Weather Prediction: A Survey," ScienceDirect. [Online]. Available: https://www.sciencedirect.com/article/pii/S2214579620300460. [Accessed: 30-Nov-2023]

[2] S. K. Panda and P. Ray, "A Survey on Weather Prediction using Big Data and Machine Learning Techniques," in Proc. 5th Int. Conf. Energy, Power, and Environment: Towards Flexible Green Energy Technologies (ICEPE), Shillong, India, 2023, pp. 1-6. doi: 10.1109/ICEPE57949.2023.10201614

[3] B. Bochenek and Z. Ustrnul, "Machine Learning in Weather Prediction and Climate Analyses—Applications and Perspectives," Atmosphere, vol. 13, no. 2, p. 180, Jan. 2022, doi: 10.3390/atmos13020180.