Application of LSTM: A Recurrent Neural Network for Temperature Prediction

Michael Zhao
The University of Texas at Dallas

Salman Jaher
The University of Texas at Dallas

Emily Nguyen
The University of Texas at Dallas

Veda Charthad
The University of Texas at Dallas

Purva Pawar
The University of Texas at Dallas

Abstract*—This paper will explore the topic of Recurrent Neural Networks (RNNs) and their applications. Specifically we will delve into the implementation of a specific variation of a RNN which is widely known as the Long Short-Term Memory. We will discuss the different applications of RNNs and how they are able to train, process, and predict sequential data. We will also compare other RNN models to the LSTM variant to demonstrate how powerful the variation is in tackling the time-series predictions compared to other baseline models. We will also discuss the mathematics and theory behind the LSTM architecture and how it is able to outperform the standard RNN in terms of better recalling the past. We will conclude with an experimental analysis on an LSTM model, predicting future temperature based on historical weather data and trends. *Note: All data used in our project was found online, publicly available on the internet.

***Index Terms*—Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), time-series data prediction**

I. Introduction

Recurrent neural networks are modified artificial neural networks made specifically for processing sequential or time-series data. RNNS can do this well because of their structure, which allows them to propagate the output of previous computations while performing the same operation for every data point in time. In contrast to artificial neural networks, the recursive connections in RNNs allow them to capture time-related dependencies in the flow.

We will focuses on a further modification of the RNN: the Long Short-Term Memory (LSTMs). The LSTM addresses the vanishing gradient problem that RNNs face. When the parameters of an RNN are updated in its backpropagation step, the gradient that gets propagated through time tends to shrink exponentially, ineffectively updating the network's weights. This becomes a problem for longer sequences and complex architectures [1]. LSTMs store the historical information of the previous sequential states in a smarter way. The effective flow of information is created by the gates in LSTMs, which regulate how much information is passed between each data point to produce the output needed. The gates play a fundamental role in capturing and forgetting the long-term data needed for large datasets and deep architectures. LSTMs are generally preferred over RNNs for their effectiveness in handling long-term dependencies and sequence variations, which outweighs the computation costs as compared to RNNs [1]. These advantages were the primary reason we used an LSTM for this project, which used a heavy dataset and a rather complex network to predict the weather for the next hour based on the data of the previous 24 hours.

II. Problem Definition

Weather and temperature forecasting can be achieved through different models such as RNNs, LSTMs, Bidirectional LSTMs (Bi-LSTMs), and Gated Recurrent Units (GRUs). Each model has different advantages and disadvantages, and some models may be better suited than others for different forecasting situations with varying time periods and datasets. We explore the effectiveness of LSTM models over other methods of regression for predicting temperature using time-series data from the Max-Planck-Insitut für Biogeochemie's Weather Station Saaleaue for the period between 2019 and 2024. An implementation of the LSTM will be created and tested using the Mean Squared Error (MSE) metric to evaluate the model.

III. Related Works

Weather data is inherently temporal, and when the issue of weather prediction is raised, most forms of neural networks are used to solve the problem at hand. Among the multitude of other papers that research weather prediction using deep learning models, the implementation of RNNs and LSTMs were frequently mentioned. A paper using RNNs for generating synthetic localized weather data proved a low error rate of 2.96 and a 185% increase in overall accuracy [3]. The study aims to provide more accurate, localized results over a longer time frame using a small set of intermittent data. The paper implements the "look-back" feature of RNN's (using previous data to build new outputs) by five days to model the prediction of the next 24 hours. The RNNs were then broken down into LSTMs and a GRU to evaluate the difference in mean squared error of the two models. The paper found that both versions of the RNN model performed significantly better than the original implementation of a feed forward neural network, despite increased computation costs [4]. While one paper debates that GRUs tend to be more effective in numerical time processing, another paper argues that LSTMs tend to be

the most effective for time-series predictions [5]. The dataset provides ten days of previous data to model the output for the next day's weather. The model's metrics as built under an LSTM show that the R^2 score of the model is 1, which proves that every variation in the predicted temperatures aligns with the actual temperatures provided in the test set. The paper compares the results of the LSTM implementation to the MapReduce Forecasting Algorithm, which creates window values in parallel and uses keys to determine the window size for determining accuracy [6]. The MapReduce algorithm produces an error value of 0.153; the LSTM model produces an error value of 0.002, which goes on to prove that recurrent neural networks and its modified models prove more accuracy and efficiency for time-series and sequential predictions.

Additionally, research for temperature forecasting using a Bhubaneswar temperature and meteorological dataset modeled and compared LSTM, Bi-LSTM, Neural Basis Expansion Analysis for Time Series (NBEATS), GRU, and Bidirectional GRU (Bi-GRU) [7]. Bi-LSTMs improves upon LSTM by adding a backward layer to the LSTM architecture. This backward layer learns from next contexts while the forward layer, which is the original LSTM structure, learns from previous contexts. The resulting architecture consists of 2 LSTM models, one for each layer. NBEATS uses stack levels which are made up of blocks that output partial forecasts. Each partial forecast from a stack level is transferred to the subsequent stack level where they are combined into a unified temperature forecast. GRU uses gates like LSTM does which control what data can pass through from previous hidden layers. The reset gate rejects certain data while the update gate facilitates the change of states. Bi-GRU expands upon GRU and uses forwards and backwards information, similar to Bi-LSTM being an expansion of LSTM. From the research experiments, LSTM had the lowest execution time which was 25.777 seconds, followed by Bi-LSTM with 32.7664 seconds, GRU with 44.0276 seconds, Bi-GRU with 46.3783 seconds, and lastly NBEATS with 769.48 seconds. The dataset used in this research was preprocessed to change data samples within a day into daily average temperatures from 2015 to 2022 in Bhubaneswar. Our implementation plans to use hourly temperature data from 2019 to 2024 to predict the 25th hour's temperature which will take much more computation since there is more data. The execution time will scale with dataset size which will result in extremely long execution times for GRU, Bi-GRU, and NBEATS. These long execution times may not be feasible for quick forecasts using more data. In terms of error performance, Bi-GRU performed the best with a Mean Squared Error (MSE) of 1.1095, followed by GRU with a MSE of 1.2442, NBEATS with a MSE of 1.2741, Bi-LSTM with a MSE of 1.2812, and lastly, LSTM that had the worst error performance with a MSE of 1.5278. It is worth noting that a better performance came at the cost of a longer execution time due to the increased complexity and variance of the model. This bias-variance tradeoff should be taken into consideration when determining which model to use for temperature forecasting.

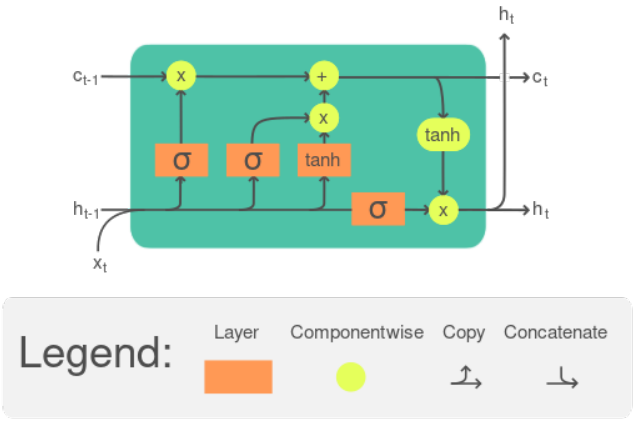

Fig. 1. Drawing of an LSTM Cell. Adapted from [8]

IV. Theoretical Technique

The Long Short-Term Memory (LSTM) network is an adapted version of the basic RNN which we have covered earlier. The primary reason why LSTMs are especially useful for sequential data because it address a major flaw in the standard RNN: the vanishing and exploding gradient problem. This problem is what makes standard RNNs perform poorly when handling long-term sequential data. This section will explore how the LSTM builds upon the RNN and is able to tackle the problems that it faces.

A. Layers of an LSTM

Like a standard RNN, an LSTM layer consists of a series of cells, each representing a timestamp. However, unlike an RNN, the LSTM cell contains a series of cells that function to better propagate the history forward and the gradient backwards. A simple drawing of the LSTM cell can be seen in Fig. 1. We will now go over each part of the LSTM and how they work.

The **forget gate** regulates how much information from the previous cell state (short-term memory) will be carried to the next state. A lower value from the forget gate leads to less of the information from being pass down.

The **input gate** determines what new information is stored in the current cell state. This is used in conjunction to the **activation gate**. By multiplying these two gates' outputs together, we get the values to add to the cell state.

The **output gate** determines what information is actually sent to the output. We take the result of the output gate and multiplies it with the current cell state to produce the output for that time step.

The **cell state** is one of the most essential components of the LSTM, as it is what gives it its long term memory. The cell state is comprised of the information from the previous cell state which is then element-wise multiplied by the output of the forget state, and then element-wise added with the output of the input gate. This is then used for the next time-step and as an input to the output gate.

These gates work together to selectively remember and process information over long sequences, allowing the LSTM to capture both short-term and long-term dependencies within the data.

B. LSTM Training Algorithm

LSTMs follow a very similar general forward and backwards propagation algorithm with gradient descent as other networks such as ANNs. LSTMs differ in their intermediate steps because of the additional gates and unrolling of through time, introducing extra complexity when doing gradient descent through the gates of each time step.

1) Forward Propagation: During forward propagation, the input data and previous cell states are passed through the gates at each time step cell. At each gate, there are different activation functions being used to determine how much of each data to retain. The output from each cell and the cell state is sent to the next time step. This process is then repeated for each time step.

2) Backward Propagation: During backward propagation, the gradient is propagated backwards from the output through every cell through the cell state. The error is split by each gate and cell state based on how their activation functions influenced the output. Gradients are calculated for each parameter of the forget gate, input gate, output gate and cell state. These gradients are used to update the weights of each gate. Once the backpropagation has reached the first timestamp, the gradients calculated at each cell is summed up and added to the respective weight.

C. Mathematical Breakdown

This section will delve into the mathematical breakdown of the forward and backward propagation steps of the LSTM to furthur and more precisely deatil how the model is trained over time. Some important notation is listed in Table I.

Symbol	Meaning
t	Time-step
h_{t-1}	Previous Hidden State
C_{t-1}	Previous Cell State
X_t	Input at time-step t
$W_{f,i,a,o}$	Input Weight for forget, input, activation, output
$U_{f,i,a,o}$	Hidden Weight for forget, input, activation, output
$b_{f,i,a,o}$	Bias for forget, input, activation, output
f_t	Output of Forget gate
i_t	Output of Input layer
a_t	Output of Activation layer
o_t	Output of Output layer

TABLE I
TABLE OF IMPORTANT SYMBOLS

1) Forward Propagation: The forward propagation is fairly straightforward. We simply follow the Forward propagation will run for T time steps, where at each time step we denote that time step as t. We iterate through time and perform the following calculations at each point:

$$
\begin{aligned}
f_t &= \sigma(W_f X_t + u_f h_{t-1} + b_f) \\
i_t &= \sigma(W_i X_t + u_t h_{t-1} + b_i) \\
a_t &= tanh(W_a X_t + u_a h_{t-1} + b_a) \\
C_t &= f_t C_{t-1} + i_t a_t \\
o_t &= \sigma(W_o X_t + u_o h_{t-1} + b_o) \\
h_t &= o_t tanh(c_t)
\end{aligned}
$$

2) Backwards Propagation: The backpropagation is admittedly a little more complex. However, we basically just follow the gradient backwards through the gates, starting with time T and working our way down to time 1.

To start, we will define the gradient on the output, δh_t:

$$\delta h_t = \frac{\partial \varepsilon_t}{\partial h_t} = \Delta t + \Delta h_t$$

We define Δt as the error produced by the current timestamp. This can be defined pretty simply by the derivative of the squared error metric. The value Δh_t is the gradient passed from the previous layer, and we will calculate this later to pass to the next cell downwards – the cell at $t = T$ should have a Δt value of 0. The calculation for Δt is shown below:

$$
\begin{aligned}
\varepsilon_t &= \frac{1}{2}(h_t - y_t)^2 \\
\Delta t &= \frac{d\varepsilon_t}{dt} = h_t - y_t
\end{aligned}
$$

The next step is to calculate the gradient for the cell state. We call this Backpropagation through Time (BPTT), as the cell state gets its gradient from the previous time steps' cell state's gradients:

$$
\begin{aligned}
\delta c_t &= \frac{\partial \varepsilon_t}{\partial t} = \frac{\partial \varepsilon_t}{\partial h_t}\frac{\partial h_t}{\partial c_t} + \frac{\partial \varepsilon_t}{\partial c_{t+1}} \\
\frac{\partial h_t}{\partial c_t} &= o_t(1 - tanh^2(c_t)) \\
\frac{\partial \varepsilon_t}{\partial c_{t+1}} &= \frac{\partial \varepsilon_t}{\partial c_{t+1}}\frac{\partial c_{t+1}}{\partial c_t} \\
&= \delta c_{t+1} * f_{t+1} \\
&= \delta h_t * o_t(1 - tanh^2(c_t)) + \delta c_{t+1} * f_{t+1}
\end{aligned}
$$

After calculating the cell state gradient, it's trivial to calculate the gradient of each of the gates. We will put the results in a matrix to make it easier to use later:

$$\delta a_t = \frac{\partial \varepsilon_t}{\partial a_t} = \frac{\partial \varepsilon_t}{\partial c_t}\frac{\partial c_t}{\partial a_t} = \delta c_t * i_t$$
$$\delta i_t = \frac{\partial \varepsilon_t}{\partial i_t} = \frac{\partial \varepsilon_t}{\partial c_t}\frac{\partial c_t}{\partial i_t} = \delta c_t * a_t$$
$$\delta f_t = \frac{\partial \varepsilon_t}{\partial f_t} = \frac{\partial \varepsilon_t}{\partial c_t}\frac{\partial c_t}{\partial f_t} = \delta c_t * c_{t-1}$$
$$\delta o_t = \frac{\partial \varepsilon_t}{\partial o_t} = \frac{\partial \varepsilon_t}{\partial h_t}\frac{\partial h_t}{\partial o_t} = \delta h_t * tanh(c_t)$$

$$\delta_{gates_t} = \frac{\partial E}{\partial gates_t} = \begin{bmatrix} \delta f_t \\ \delta i_t \\ \delta a_t \\ \delta o_t \end{bmatrix} = \begin{bmatrix} \delta c_t * c_{t-1} \\ \delta c_t * a_t \\ \delta c_t * i_t \\ \delta h_t * tanh(c_t) \end{bmatrix}$$

We need to calculate the gradients of each weight with respect to the gates. This would be a lot of manual math to write out. Instead, notice that the partial derivative of each of the weights (input weight, hidden weight, and bias weight) are only off by a constant term – x_t, h_{t-1}, and 1 respectively. Instead, we simply define:

$$\delta gates_t{}^* = \frac{\partial gates_t}{\partial b} = \begin{bmatrix} f_t(1-f_t) \\ i_t(1-i_t) \\ 1-a_t^2 \\ o_t(1-o_t) \end{bmatrix}$$
$$\delta W_t{}^{(gates)} = \frac{\partial gates_t}{\partial W} = \delta gates_t{}^* * X_t$$
$$\delta U_t{}^{(gates)} = \frac{\partial gates_t}{\partial U} = \delta gates_t{}^* * h_{t-1}$$
$$\delta b_t{}^{(gates)} = \frac{\partial gates_t}{\partial b} = \delta gates_t{}^*$$

Given these values, we can now calculate the gradient that we will pass to the previous time step layer:

$$\begin{aligned} \Delta h_{t-1} &= \frac{\partial \varepsilon_t}{\partial h_{t-1}} = \frac{\partial \varepsilon_t}{\partial gates_t}\frac{\partial gates_t}{\partial h_{t-1}} \\ &= (\delta gates_t)^T \odot \delta_{h_{t-1}}^{gates_t} \\ \delta_{h_{t-1}}^{gates_t} &= \frac{\partial gates_t}{\partial h_{t-1}} \\ &= U^T \odot \delta gates_t^* \end{aligned}$$

The final step in this cell is to compute the gradients with respect to each weight:

$$\delta W_t = \frac{\partial \varepsilon_t}{\partial W} = \frac{\partial \varepsilon_t}{\partial gates_t}\frac{\partial gates_t}{\partial W} = \delta gates_t \odot \delta_{W_t}^{gates}$$
$$\delta U_t = \frac{\partial \varepsilon_t}{\partial U} = \frac{\partial \varepsilon_t}{\partial gates_t}\frac{\partial gates_t}{\partial U} = \delta gates_t \odot \delta_{U_t}^{gates}$$
$$\delta b_t = \frac{\partial \varepsilon_t}{\partial b} = \frac{\partial \varepsilon_t}{\partial gates_t}\frac{\partial gates_t}{\partial b} = \delta gates_t \odot \delta_{b_t}^{gates}$$

Symbol	Unit	Variable
Date Time	EU Std. Format	Date and time of the data record (end)
p	mbar	air pressure
T	^oC	air temperature
Tpot	K	potential temperature
Tdew	^oC	dew point temperature
rh	%	relative humidity
VPmax	mbar	saturation water vapor pressure
VPact	mbar	actual water vapor pressure
VPdef	mbar	water vapor pressure deficit
sh	$g\ kg^{-1}$	specific humidity
H2OC	mmol mol^{-1}	water vapor concentration
rho	$g\ m^{-3}$	air density
wv	ms^{-1}	wind velocity
max. wv	ms^{-1}	maximum wind velocity
wd	0	wind direction
rain	mm	precipitation
raining	s	duration of precipitation
SWDR	Wm^{-2}	short wave downward radiation
PAR	$\mu mol\ m^{-2}s^{-1}$	photosynthetically active radiation
max. PAR	$\mu mol\ m^{-2}s^{-1}$	maximum photosynthetically active radiation
Tlog	^oC	internal logger temperature
CO2	ppm	CO_2 concentration of ambient air

TABLE II
DATASET COLUMN DESCRIPTIONS. ADAPTED FROM [9].

3) Final Weight Updates: At the end of the training cycle or when T reaches its maximum value, the weights will have a singular final update which is equivalent to the summed deltas of each time step:

$$\delta W = \sum_{t=0}^{T} \delta W_t \longrightarrow W_{new} = W_{old} - \eta \delta W$$
$$\delta U = \sum_{t=1}^{T} \delta U_t \longrightarrow U_{new} = U_{old} - \eta \delta U$$
$$\delta b = \sum_{t=0}^{T} \delta b_t \longrightarrow b_{new} = b_{old} - \eta \delta b$$

V. IMPLEMENTATION

Our model was implemented in Python, using only a handful of libraries. The main libraries used in the model was pandas for DataFrame management, numpy for its excellent NDArray structure, and matplotlib to generate the output graphs.

A. Dataset Analysis and Preprocessing

The dataset we utilized was the weather data from the Max-Planck-Insitut fuer Biogeochemie – Wetterdaten, which provided 10-minute interval weather data from 2003 to present [9]. This dataset provided us with a multitude of attributes, listed in Table II. Additionally, this dataset is completely free and on their website for anyone to use! One issue we faced at many stages was the fact that there was an overwhelming amount of data. Before we even started to build the model, we needed to clean up the data and reduce it to a manageable amount.

The first thing we did was limit how much data we used, as more than 2 decades of data was excessive. We settled on 5 years of data, opting for the more recent 2019-2023 data. Once pulling all the data into a single DataFrame, we moved onto cleaning the data. Luckily, it didn't seem like the dataset contained any erroneous values and all the standard deviations fell within reasonable margins. We broke down the individual components of the data and did some correlation analysis. We found which components best correlated with the output data and isolated those, performing a simple dimensionality reduction. The correlation can be seen in Table III.

After the reduction, we were left with a total of 7 input columns: temperature, saturation air pressure, air density, vapor pressure deficit, water vapor concentration, vapor pressure, and specific humidity. This cuts down from the original 22 columns and heavily reduces model complexity.

Column	Correlation
Tpot (K)	0.99570745904994
Tlog (degC)	0.9774599168887204
VPmax (mbar)	0.9610091323707131
rho (g/m^3)	0.9510897177905429
Tdew (degC)	0.8239077954828616
VPact (mbar)	0.8022935558449872
H2OC (mmol/mol)	0.8015619417624299
sh (g/kg)	0.8011715337780086
VPdef (mbar)	0.7695248633087297
rh (%)	0.5849969513274151
PAR (µmol/m²/s)	0.472126454413622
SWDR (W/m²)	0.4591117772891701
max. PAR (µmol/m²/s)	0.4353262850629407
max. wv (m/s)	0.11889334102190578
raining (s)	0.10790655742057835
p (mbar)	0.07178334941392706
wd (deg)	0.040054995539300434
CO2 (ppm)	0.024596290708999245
wv (m/s)	0.009173111020638914
rain (mm)	0.007558429670351667

TABLE III
CORRELATION OF INPUT DIMENSIONS TO TEMPERATURE

Another key reduction we made to our data size was downsampling the data from a 10-minute interval to 60 minute intervals. Because we assumed that the temperature couldn't change that much in the span of an hour, we opted to simply keep 1 in every 6 data points. This still left us with 40,000 unique data points.

We chose a 70% / 20% / 10% training, validation, and test split, respectively. This allowed us to use 30,000 training points with 8,000 validation points. This was ideally the amount of data that we could use. However, we later realized that on our computers this was still too much to train in a reasonable amount of time and further reduced each one of the sets by 80%. One thing to note is that for this train/test split, we were NOT able to randomize the data or select randomly distributed data as this is time-series data. That meant that the data needed to retain its sequential order. Therefore, we did a simple cut of the data, where the first continuous 70% of the data was put into the training array and so on with the other two.

Learning Rate	Train MSE	Validation MSE	Overfitting Point
0.1	0.0137	0.3013	2 epochs
0.01	0.0115	0.0318	9 epochs
0.001	0.0185	0.0169	N/A
0.005	0.0132	0.0140	16 epochs

TABLE IV
MSE VALUES FOR DIFFERENT LEARNING RATES

To train such a model, we needed to slice our data into time segments. We will be training our LSTM model to take the data from the past n hours and predict the next hour. This means that each "slice" will consist of n input and output points, with the output points offset by 1 hour forward from the data (to simulate predicting the next hour with the past data). These slices made it much easier to feed data into our model as each test point would be a set of n hours of data.

B. Data Structures for LSTM

For the actual LSTM, we needed to implement a couple of data structures to hold the required information. We define n to be the number of input dimensions and T to be the total number of time steps we use to predict the next time step. The weights were stored in a $4 \times 3 \times n$ matrix, where the axis represented the gates (f, i, a, o), different weights (w, u, b), and the input dimensions, respectively. For example, the hidden weight of the activation gate of the 4th input dimension would be $U_f^{(0)} = weights[0, 1, 3]$. Note that this differs from the math, where we assumed the inputs would be a scalar and not a vector.

The outputs are stored in a $6 \times T \times n$ matrix, representing the gate/main outputs (f, i, a, o, c, h), each timestamp, and each input dimension, respectively. The output of the LSTM layer is connected to a dense layer, with n inputs and 1 output. This output requires weights on each one of its input $\rightarrow$ output connections and a bias on the output node itself. These are stored in a vector and scalar variable respectively.

C. Training Details

The weights are initialized according to Xavier Initialization. This is to reduce any bias that could be introduced by more naive methods of initialization such as raw uniform or normal distributions. Given n is the size of the inputs and T is the total number of time steps, the formula is as follows:

$$w_i = \sqrt{\frac{6}{n+T}}$$

The model is then implemented according to the math. Tests are run in epochs, each of which runs through the entire training set followed by the validation set (only forward propagation) and prints the loss for both. Following training, the test set is run through the model, and we plot the training loss as well as the graph of actual vs predicted temperatures.

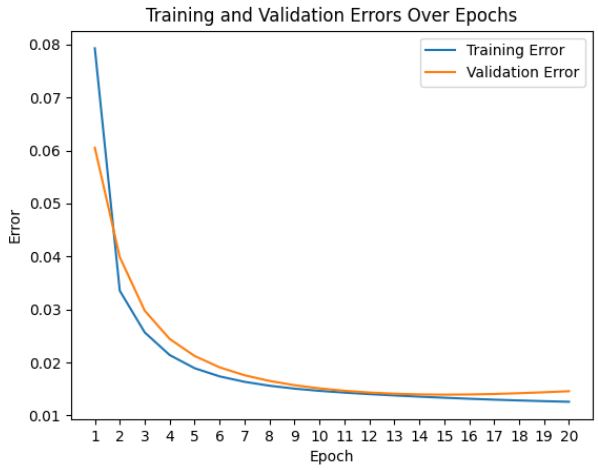

Fig. 2. Training loss of LSTM Model

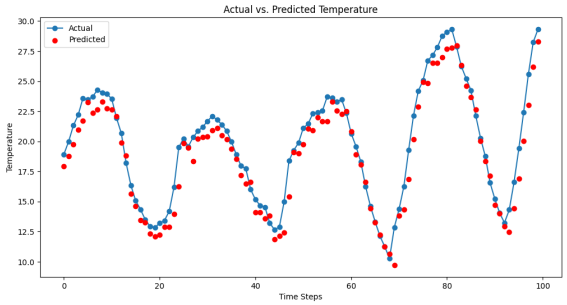

Fig. 3. Actual vs Predicted Temperature

VI. Results and Discussion

Since the model took so long to run, we focused our experimentation on the most important factor for our model, learning rate. By changing the learning rate, we can determine how well the LSTM trains. We found that the model is pretty sensitive to learning rate, and many values quickly overfit the model. This can be seen in IV. The best value for the learning rate ended up being 0.005, with higher learning rates overfitting too fast and lower learning rates being too slow. This could be partially due to the fact that we had to severely cut down our dataset due to limited computing resources we had when training.

Even with the optimized learning rate, we found that after 14 epochs, the model will start to overfit. This behavior can be seen in Fig. 2. For the test data, we got around the same error: $MSE = 0.018827457765246185$. This was pretty good and improved by the test MSE gotten from our baseline model.

Our baseline model was a simple model that would just use the previous time stamp's temperature to predict the future temperature. The baseline model performed decently well when tested with the test set, outputting $MSE = 0.02123422215523$. We've shown that our LSTM model performs better than the baseline. However, it doesn't perform that much better, and we believe that is heavily attributed to the limited computation power, leading to a limited number of training examples we could use. Additionally, since we were manually writing all of the model from scratch, we didn't want to add too much complexity as it would lead to the increase in bugs. This would result in an unfinished project as we did not have as much time as we hoped to have.

Visually, the temperature predictions are fairly accurate, with some predictions getting the temperature exactly correct. An interesting trend to note is that the predictions are much more accurate when the temperature drops as opposed to when the temperature suddenly increases. The model also struggled a lot with the higher end of the temperature scale. We believe that may be from some accidental bias introduced or just an imbalance in high vs low temperatures in the subset of data we took. These predicted data points against the actual temperature can be seen in Fig. 3.

A. Note on Past Iterations and Failures

Originally, we were planning on doing a multi-class classification with weather conditions – rainy, sunny, etc. However, that dataset presented way too many issues, and we decided to focus on a simpler dataset due to the time constraints and the fact that we had to code all of this from scratch.

To start, the dataset itself was too small and had a class imbalance issue. The size of the dataset limited our training but was still doable. However, the class imbalance issue provided a much more difficult challenge to tackle. We were thinking about weighting each one of the classes based on how common each class was. In this dataset, it was sunny 40% of the time, rainy 40% of the time, cloudy 10% of the time, and all the other 3 classes that last 10% of the time. That means that our model basically only predicted it was either sunny or rainy. However, weighting classes came with its own problems, such as overfitting, and we would need to add additional regularization parameters, which we did not have time for.

Additionally, the complexity of the model came from the multi-class classification. In addition to the regularization we would need to add, we also probably needed multiple layers of the LSTM to accurately get a good result. We were using a dense, softmax layer, which just created a longer training session. In the end, we weren't able to get good results and scrapped the idea for this simpler temperature prediction problem.

VII. Conclusion and Future Work

Our current model is able to take in the various data features from the previous day and predict the temperature for the upcoming day fairly accurately. We noticed some of the variation that was produced by the limited dataset, the limited computing power, and untested settings such as the number of previous days to use. Our model still performs better than the baseline model, despite the listed limitations.

The current implementation of the LSTM model takes in many attributes as the input including air temperature, air pressure, and humidity to predict the temperature. However, a

future implementation on this project would be to predict more features from the trained data. Some possible features that we might be able to predict would be the future humidity and the weather conditions for the upcoming days. Another consideration for the future work would be to improve the quality of the results we output and improve the accuracy of our model. With the use of additional training data and hyperparameter tuning, we should be able to optimize the model to provide better predictions for the temperature predictions.

REFERENCES

[1] J. Ma, Y. Li, F. Ma, J. Wang, and W. Sun, "A comparative study on the influence of different prediction models on the performance of residual-based monitoring methods," Computer Aided Chemical Engineering, pp. 1063–1068, 2022. doi:10.1016/b978-0-323-95879-0.50178-8

[2] Greff, Klaus, et al. "LSTM: A search space odyssey." IEEE transactions on neural networks and learning systems 28.10 (2016): 2222-2232.

[3] J. M. Han, Y. Q. Ang, A. Malkawi, and H. W. Samuelson, "Using recurrent neural networks for localized weather prediction with combined use of public airport data and on-site measurements," Building and Environment, vol. 192, p. 107601, Apr. 2021. doi:10.1016/j.buildenv.2021.107601

[4] J. Chung, C. Gulcehre, K. Cho, Y. Bengio, Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, in: NIPS 2014 Deep Learning and Representation Learning, 2014.

[5] A. Srivastava and A. S, "Weather Prediction Using LSTM Neural Networks," 2022 IEEE 7th International conference for Convergence in Technology (I2CT), Mumbai, India, 2022, pp. 1-4, doi: 10.1109/I2CT54291.2022.9824268.

[6] A. Sinhaa and P. Janaa, "MRF: MapReduce based Forecasting Algorithm for Time Series", ScienceDirect: India, 2018.

[7] N. K. Sabat, R. Nayak, U. C. Pati and S. Kumar Das, "A Comparative Analysis of Univariate Deep Learning-based Time-series Models for Temperature Forecasting of the Bhubaneshwar," 2022 IEEE 2nd International Symposium on Sustainable Energy, Signal Processing and Cyber Security (iSSSC), Gunupur, Odisha, India, 2022, pp. 1-5, doi: 10.1109/iSSSC56467.2022.10051494

[8] O. V. G. C. ketograff Redrawn as SVG by, "Structure of a LSTM (Long Short-term Memory) cell." Wikimedia Commons, Sep. 23, 2020. https://commons.wikimedia.org/wiki/File:LSTM_cell.svg

[9] "Max-Planck-Institut fuer Biogeochemie – Wetterdaten," www.bgc-jena.mpg.de. https://www.bgc-jena.mpg.de/wetter/