Time Series Analysis and Forecasting

Tirth Chaudhari
Department of Mathematics and Computing,
Dhirubhai Ambani University

Shreya Patel
Department of Information and Communication Technology,
Dhirubhai Ambani University

April 2025

Abstract

Forecasting Daily Temperature in NYC Central Park (1869–2022):

This report explores over 150 years of daily temperature data from Central Park, NYC, to identify trends, seasonality, and forecast future temperatures. Using statistical time series analysis, particularly ARIMA-based models, the study evaluates the performance of autoregressive, moving average, and integrated models. The analysis reveals strong seasonal patterns and a slight long-term warming trend, with ARIMA-based models delivering the most accurate forecasts. Key learnings include insights into stationarity, random walk behavior, and residual analysis for model validation.

Problem Statement

The goal of this analysis is to forecast future temperatures in New York City using historical weather data recorded at Central Park from 1869 to 2022. With evidence of climate change and rising temperatures, it's crucial to understand temperature dynamics and improve forecasting models. The primary challenge is to model a long, complex, seasonal time series and evaluate different approaches—AR, MA, ARMA, ARIMA and SARIMA —based on their forecasting accuracy.

Dataset description

The dataset is sourced from the National Weather Service, which has maintained daily weather records at Central Park since 1869. The dataset includes the following key features:

- **DATE**: This is the timestamp for each observation, recorded in the format `YYYY-MM-DD`. It serves as the index for time series modeling and allows for chronological ordering, resampling, and rolling operations.
- **PRCP (Precipitation)**: Measured in inches, this feature represents the total precipitation (including rain and melted snow) recorded on that particular day. This is useful for broader weather trend analysis but was not directly used in the temperature forecasting model.
- **SNOW (Snowfall)**: Also measured in inches, this indicates the amount of new snowfall recorded during a 24-hour period. Like PRCP, this was not a target variable in our study but provides useful insight into seasonality and extreme weather conditions.
- **SNWD (Snow Depth)**: This feature reflects the depth of accumulated snow on the ground, measured in inches. However, it was observed that SNWD has significant missing data, especially prior to the year 1912, when routine measurement of snow depth was less standardized. Due to the high proportion of missing values and limited correlation with our target variable (`TEMP`), this column was dropped entirely during preprocessing to avoid introducing bias or unnecessary complexity.
- **TMIN (Minimum Temperature)**: The lowest temperature recorded on each day, in degrees Fahrenheit. This variable is one of the two components used to estimate the overall daily temperature.
- **TMAX (Maximum Temperature)**: The highest temperature recorded on each day, also in degrees Fahrenheit. Along with TMIN, it helps us estimate the daily mean temperature, which serves as the main target variable for this forecasting task.

To simplify the modeling task and provide a smoothed representation of daily weather, a new feature was created:

- **TEMP (Average Daily Temperature)**:

The TEMP variable is computed as the average of TMIN and TMAX:

$$\text{TEMP} = \frac{\text{TMIN} + \text{TMAX}}{2}$$

Here we average TMIN and TMAX to provide a better representation of the full day's temperature pattern.

Time Series Forecasting

1. Data Preprocessing

Step 1: Convert the `DATE` Column to Datetime Format and Set It as the Index

Step 2: Handle Missing Values

- **A. `TMIN` and `TMAX`: Filled with Median Values**
  - These columns represent the daily minimum and maximum temperatures, which are essential for modeling the daily average temperature.
  - **Why the median?**
    - The median is more robust to outliers compared to the mean and is particularly effective for skewed distributions, which is often the case with temperature data.
- **B. `SNOW`: Filled with 0**
  - The `SNOW` column, representing daily snowfall, contains numerous missing values, particularly outside of winter months.
  - **Assumption:**
    - If snowfall data was not recorded, we assume that no snowfall occurred on that day.
- **C. `SNWD`: Dropped Entirely**
  - The `SNWD` (Snow Depth) column had 16,558 missing values, particularly before 1912.
  - Given that imputation would require speculative assumptions, and since snow depth data was sparse throughout most of the dataset, the best approach was to drop this column entirely.

Step 3: Create a New Column `TEMP` (Average Daily Temperature)

- **Why create this column?**
  - The average daily temperature provides a smoothed, central measure of the daily temperature behavior, making it useful for climate modeling.
  - It also helps to reduce the noise between the `TMIN` and `TMAX` columns.

2. Visualization and Initial Observations

Once the dataset is cleaned and the `TEMP` column has been created, the next step is to visually explore the time series in order to understand its underlying structure before applying any models. This analysis includes identifying patterns such as trends, seasonality, noise, and any potential anomalies.

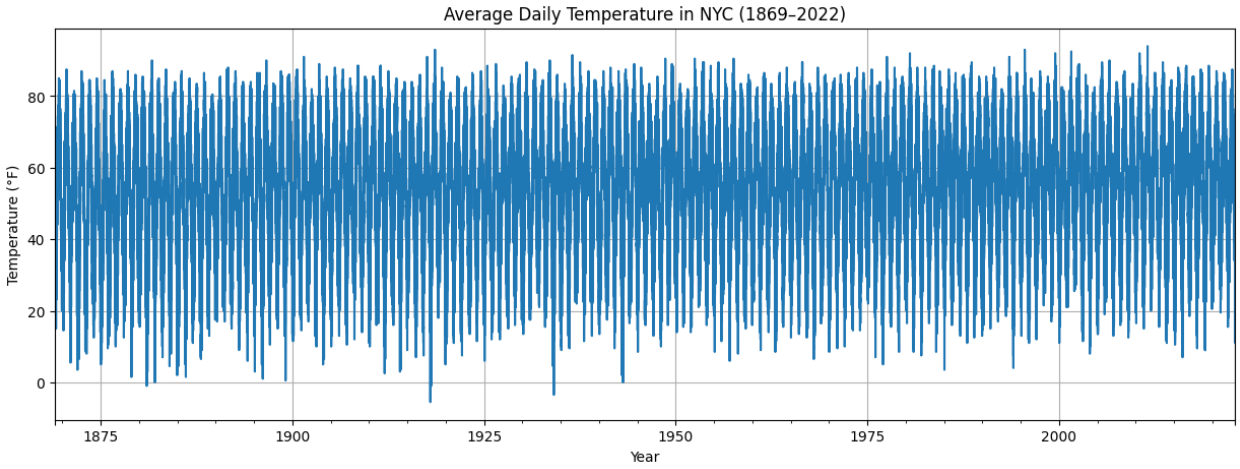

Figure 1: Time series plot of daily average temperature.

What do we observe in the time series plots?

The plot reveals distinct cyclical patterns, characterized by sharp and repetitive increases and decreases in temperature every year. These fluctuations are due to the natural seasonal climate cycle, with hot summers and cold winters. There subtle upward shift in the upper bounds of the temperature range in recent years which suggests a potential long-term warming trend.

Does the data exhibit any trend or seasonality?

- **Seasonality:** There is a clear and consistent yearly seasonal pattern. Each year, temperatures rise and fall in a predictable wave-like manner, with the peak typically occurring in summer and the trough in winter.
- **Trend:** While the seasonal variation dominates, there is a faint upward trend in temperature over the course of the century, which may indicate the impact of climate change. This trend is not immediately prominent due to the scale of the seasonal fluctuations.

3. Stationarity Testing

Why Test for Stationarity?

In time series forecasting, testing for stationarity is essential because many statistical models—such as Autoregressive (AR), Moving Average (MA), Autoregressive Integrated Moving Average (ARIMA), and Seasonal ARIMA (SARIMA)—assume that the time series is stationary. A stationary series is one whose statistical properties, such as mean, variance, and autocorrelation, remain constant over time.

If the time series is non-stationary, model assumptions may be violated, resulting in inaccurate forecasts and poorly behaved residuals.

Augmented Dickey-Fuller (ADF) Test

The Augmented Dickey-Fuller (ADF) test is commonly used to assess stationarity by checking for the presence of a unit root.

Hypotheses of the ADF Test:

- Null Hypothesis (H_0): The time series has a unit root — i.e., it is non-stationary.
- Alternative Hypothesis (H_1): The time series does not have a unit root — i.e., it is stationary.

Test Results:

- ADF Statistic: `-26.771843878064434`
- p-value: `0.0`

Conclusion:

Since the p-value is below the commonly used threshold of 0.05, we reject the null hypothesis in favor of the alternative. This strongly indicates that the time series does not contain a unit root and is therefore stationary.

Thus, no differencing is required for this dataset, and it is ready for modeling with stationary-assuming methods.

So Why Still Use ARIMA, Which Assumes Differencing?

Although the Augmented Dickey-Fuller (ADF) test indicates that the time series is stationary, there may still be underlying seasonal patterns or structural changes that could benefit from differencing. For this reason, an ARIMA model with one level of differencing—specifically, ARIMA(2,1,1)—was also tested.

Applying differencing in this case serves as a diagnostic step: it allows for a comparison between models to evaluate whether differencing yields better forecasting accuracy, even when stationarity appears to be present.

4. Random Walk Behavior

According to the Augmented Dickey-Fuller (ADF) test, the temperature time series is stationary, with an ADF statistic that is significantly negative and a p-value zero. However, while this confirms stationarity, it does not fully define the appropriate model structure for forecasting. This is where the ACF (Autocorrelation Function) and PACF (Partial Autocorrelation Function) come into play.

1. Stationarity Does Not Imply Model Order

The ADF test specifically checks for the presence of a unit root—essentially verifying whether the series follows a random walk. It does not capture the dynamics of autocorrelation within the data. Even a stationary time series may exhibit significant autocorrelations, which are key to determining AR (autoregressive) and MA (moving average) terms.

2. ACF and PACF Guide Model Selection

The ACF and PACF plots are essential tools in identifying the parameters for ARMA or ARIMA models:

- **PACF:** Helps determine the number of AR (autoregressive) terms, denoted by p.
- **ACF:** Helps determine the number of MA (moving average) terms, denoted by q.

Even when the series is stationary, these plots are necessary to estimate the correct model structure.

3. Detecting Seasonality with ACF/PACF

The ADF test does not detect seasonal patterns. However, the ACF and PACF plots often exhibit distinct seasonal spikes. In this dataset, strong spikes at lag 365 indicate a yearly seasonal pattern—critical for constructing seasonal models such as SARIMA.

What Is a Random Walk? A random walk is a common form of non-stationary time series, defined as:

$$Y_t = Y_{t-1} + \varepsilon_t$$

where ε_t is white noise (i.e., a random, zero-mean error term).

Properties of a Random Walk:

- No mean-reversion (the series does not stabilize around a central value).
- No deterministic trend—only past value matters.
- Increasing variance over time; the series tends to wander.

Original Series Analysis Using ACF and PACF

ACF (Autocorrelation Function)

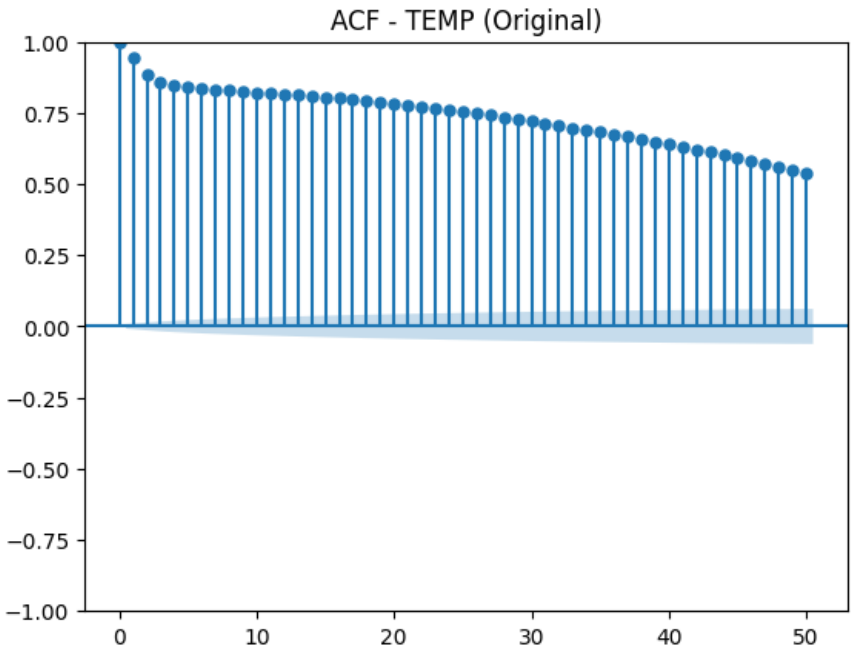

Figure 2: Autocorrelation Function (ACF) of daily average temperature.

Here the ACF shows:

- Strong autocorrelations that decay slowly over time, indicating persistent dependencies.

Such patterns are typical of temperature data, which often follows consistent seasonal cycles due to annual climatic variation.

PACF (Partial Autocorrelation Function)

And PACF shows:

- Significant spikes at lower lags (e.g., 1, 2), suggesting that the temperature of a given day is heavily influenced by the temperatures of the preceding few days.

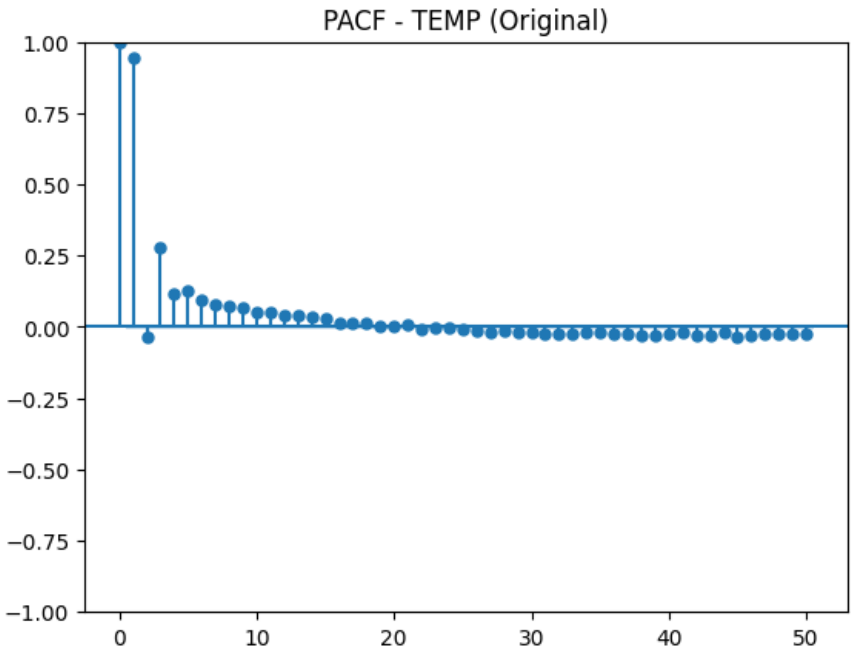

Figure 3: Time series plot of daily average temperature.

This indicates that not only do recent observations impact current values, but there is also a relationship with values from the same period in previous years.

Interpretation
The combination of short-term and seasonal autocorrelation implies a structure where:

- Short-term lags (1–3) capture recent memory and local dynamics.

Conclusion
The original temperature time series demonstrates:

- Clear evidence of both short-term and long-term (seasonal) dependencies.
- Strong seasonal structure typical of meteorological data.
- Stationarity, as confirmed by the ADF test — implying no differencing is needed.

Hence, the series is *stationary but seasonal*, making it well-suited for models such as SARIMA, which can capture both autoregressive and seasonal components.

5. Forecasting Models and Evaluation

To forecast future temperature values, five time series models were evaluated using the last five years of historical data. Each model was assessed using diagnostic tools such as residual plots and Q-Q plots.

Model Diagnostics:

- **Residual Plot:** Ideally, residuals should resemble white noise — randomly scattered around zero with no discernible pattern or trend.

  - Randomly scattered around zero → good model fit.
  - Patterns or trends → indicates potential model misspecification.

- **Q-Q Plot:** Compares the distribution of residuals against a normal distribution.
  - Points align along the 45° line → residuals are approximately normally distributed.
  - Deviations indicate skewness or outliers.

AutoRegressive (AR) Model — ARIMA(2, 0, 0)

Model Description:

This is a purely autoregressive model where:

- **p = 2**: The model uses the previous two observations to predict the current value.
- **d = 0**: No differencing is required as the data is already stationary.
- **q = 0**: No moving average terms are included.

The model assumes today's temperature depends linearly on the temperatures of the previous two days.

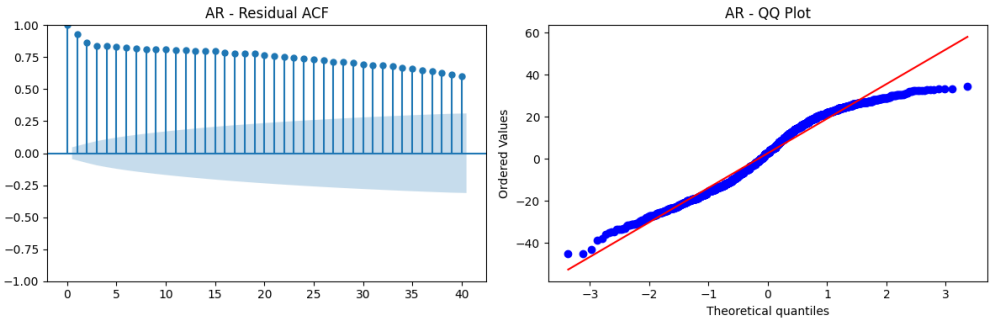

Figure 4: Residual and Q-Q Plots for AR(2) Model

Moving Average (MA) Model — ARIMA(0, 0, 1)

Model Description:

This is a moving average model of order 1:

- **p = 0**: No autoregressive terms.
- **d = 0**: Data is stationary.
- **q = 1**: Incorporates one lag of forecast error to predict current value.

Here, the prediction is based on how inaccurate the last forecast was.

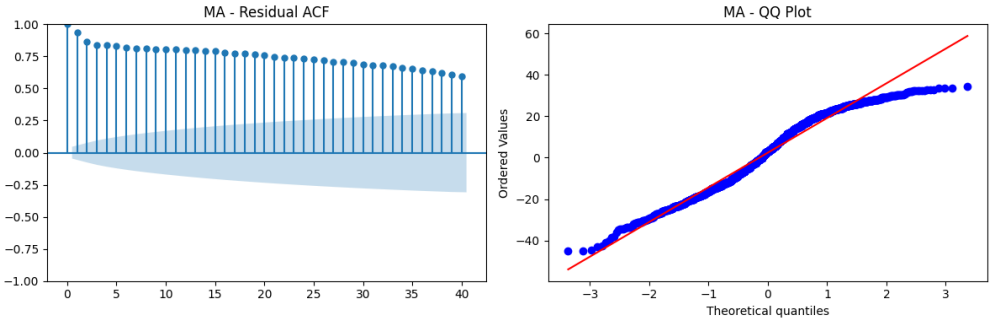

Figure 5: Residual and Q-Q Plots for MA(1) Model

ARMA Model — ARIMA(2, 0, 1)

Model Description:

Combines both autoregressive and moving average components:

- **p = 2**: Utilizes the last two observed values.
- **d = 0**: No differencing; assumes data is stationary.
- **q = 1**: Includes one lag of forecast error.

This model captures both past values and error structure.

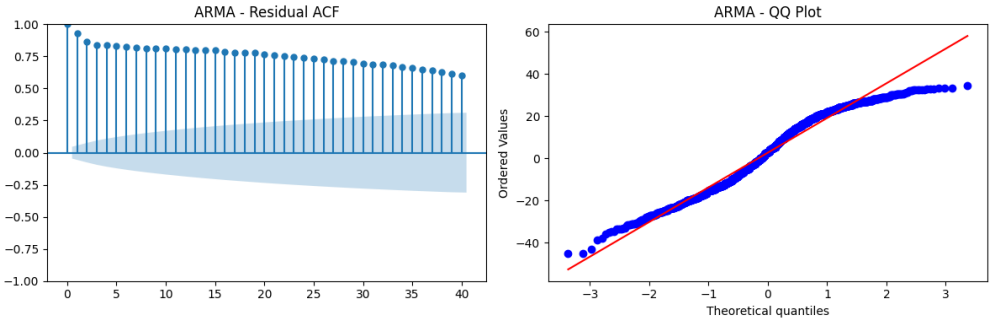

Figure 6: Residual and Q-Q Plots for ARMA(2,1) Model

ARIMA Model — ARIMA(2, 1, 1)

Model Description:

Adds one level of differencing:

- **p = 2**: Two lagged values of the differenced series.
- **d = 1**: First-order differencing applied to handle potential trend.
- **q = 1**: One lag of forecast error included.

While the series was considered stationary, this model was included for comparative purposes, as differencing can sometimes enhance performance.

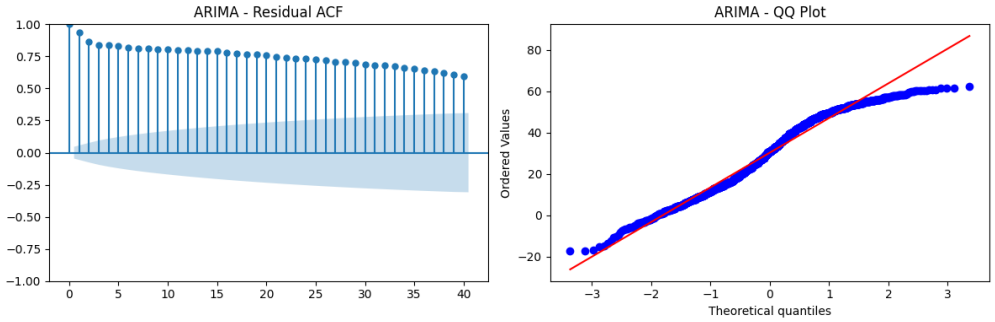

Figure 7: Residual and Q-Q Plots for ARIMA(2,1,1) Model

Seasonal ARIMA (SARIMA) Model — SARIMA(2,1,1)(1,1,1,52)

Model Description:

This model adds a seasonal component to the non-seasonal ARIMA structure:

- **Non-seasonal:** AR(2), I(1), MA(1)
- **Seasonal:** SAR(1), SI(1), SMA(1) with a period of 52 (weekly seasonality)

Applied to weekly-averaged temperature data, this model captures both short-term dynamics and annual seasonal patterns.

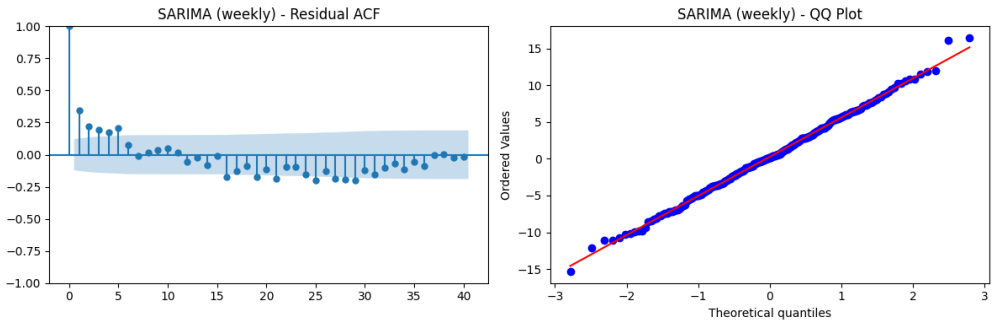

Figure 8: Residual and Q-Q Plots for SARIMA(2,1,1)(1,1,1,52) Model

Also, to assess the performance of each forecasting model, two standard evaluation metrics were used:

- **Mean Squared Error (MSE):** Measures the average of the squared differences between predicted and actual values. It penalizes larger errors more heavily.
- **Mean Absolute Error (MAE):** Represents the average of the absolute differences between predicted and actual values, offering a more interpretable measure of average model error.

The table below summarizes the MSE and MAE for each model:

Model	MSE	MAE
SARIMA (weekly)	28.07	4.18
AR	285.27	14.56
ARMA	285.59	14.57
MA	294.49	14.79
ARIMA	1205.68	30.58

Table 1: Forecasting Performance of Time Series Models

Overall Insight

The SARIMA model significantly outperformed all other models, demonstrating its ability to effectively capture the structure of the data. Its superior performance can be attributed to the following:

- **Seasonal Adaptation:** SARIMA explicitly models regular cyclic patterns (e.g., weekly and yearly temperature fluctuations), which are present in the data.
- **Stationary Foundation:** The underlying series is stationary with a seasonal component. SARIMA accommodates this with both seasonal and non-seasonal terms.
- **Model Fit:** Simpler models such as AR, MA, and ARMA capture only short-term dependencies and fail to address the broader seasonal trends.

In summary, SARIMA provides the best balance between capturing local patterns and modeling seasonality, making it the most suitable forecasting model for this temperature data.

6. Key Learnings

Through the process of modeling and evaluating temperature time series data, several important insights were gained:

- **Seasonality is dominant:** Temperature data exhibits strong and regular seasonal patterns. Non-seasonal models such as AR, MA, or even ARIMA without seasonal terms are insufficient for capturing this structure.
- **Effectiveness of SARIMA:** When seasonality is clear and periodic, SARIMA models perform exceptionally well. They are particularly well-suited for long-term climate data, where both short-term dynamics and recurring patterns are present.
- **Benefits of Data Aggregation:** Aggregating data (e.g., computing weekly averages) helps smooth out high-frequency noise, improving the signal-to-noise ratio and enabling models to better detect seasonal and trend components.
- **Importance of Stationarity Testing:** Preliminary tests such as the Augmented Dickey-Fuller (ADF) test, along with autocorrelation (ACF) and partial autocorrelation (PACF) plots, are critical for identifying the appropriate level of differencing and guiding model selection.
- **Value of Historical Data:** Historical weather records can reveal broader climate trends, such as gradual warming patterns, which are particularly evident in urban environments like New York City.

Time Series Forecasting using Deep Learning

1 Univariate One-Step Forecasting

Motivation and Objective

The goal of this section is to forecast the next day's maximum temperature (`TMAX`) using only historical values of `TMAX`. Traditional forecasting methods such as ARIMA or exponential smoothing often rely on strong assumptions of linearity and stationarity, which are not always valid in real-world weather systems. In contrast, deep learning models, particularly recurrent architectures, can model complex, nonlinear temporal dependencies without requiring extensive manual feature engineering. Therefore, we adopt deep neural networks such as RNN, LSTM, and GRU for this forecasting task.

Data Preparation and Modeling Strategy

For this univariate task, we use past n days of `TMAX` to predict the next day's value. The data is transformed into a supervised learning problem using a fixed-length sliding window approach. Given a window size w, the input and target are defined as:

$$X^{(t)} = [TMAX_{t-w}, TMAX_{t-w+1}, \ldots, TMAX_{t-1}], \quad y^{(t)} = TMAX_t$$

Normalization

All temperature values were scaled to the $[0, 1]$ range using MinMaxScaler:

$$\tilde{TMAX} = \frac{TMAX - T_{\min}}{T_{\max} - T_{\min}}$$

Chronological Splitting of Time Series Data

Unlike typical i.i.d. datasets, time series data exhibits strong temporal dependencies, so random shuffling is not appropriate. Instead, we divide the data **chronologically** to preserve the sequential nature of the observations.

Let the total number of data points after preprocessing be N. We partition the data into three subsets:

- **Training set**: the first 70% of the data, used to learn model parameters.
- **Validation set**: the next 20%, used to tune hyperparameters and prevent overfitting.
- **Test set**: the final 10%, used for evaluating the generalization performance of the trained model.

Let the window size be w and the total available samples after windowing be $M = N - w$. Then:

$$\begin{aligned} M_{\text{train}} &= \lfloor 0.7 \cdot M \rfloor \\ M_{\text{val}} &= \lfloor 0.2 \cdot M \rfloor \\ M_{\text{test}} &= M - M_{\text{train}} - M_{\text{val}} \end{aligned}$$

This ensures there is no temporal leakage across subsets. The test set represents the most recent and unseen portion of the time series.

One-Step Forecasting Strategy

For one-step univariate forecasting, we use a sliding window approach. A fixed number of past observations (e.g., 30 days) are used to predict the temperature of the following day. Let w denote the window size. Then, from a time series of length N, we construct $M = N - w$ training examples as follows:

$$X_i = [x_i, x_{i+1}, \ldots, x_{i+w-1}], \quad y_i = x_{i+w}$$

This sliding window moves one step forward each time, ensuring each target y_i corresponds to the next immediate day following the input window X_i.

Chronological Windowing Example

Suppose we have a univariate time series of daily maximum temperatures over 100 days:

$$\text{TMAX series: } x_1, x_2, x_3, \ldots, x_{100}$$

We choose a window size of $w = 30$, which means each training sample consists of 30 consecutive days used to predict the next day's temperature. That is, for each i:

$$X_i = [x_i, x_{i+1}, \ldots, x_{i+29}], \quad y_i = x_{i+30}$$

This gives $M = 100 - 30 = 70$ total samples. For example:

- Sample 1: $X_1 = [x_1, x_2, \ldots, x_{30}]$, $y_1 = x_{31}$
- Sample 2: $X_2 = [x_2, x_3, \ldots, x_{31}]$, $y_2 = x_{32}$
- $\vdots$
- Sample 70: $X_{70} = [x_{70}, x_{71}, \ldots, x_{99}]$, $y_{70} = x_{100}$

We then split the dataset chronologically:

- First 70% (49 samples) for training
- Next 20% (14 samples) for validation
- Final 10% (7 samples) for testing

This ensures the temporal integrity of the time series is preserved, and that future values are never used to predict past data.

This sliding window transformation ensures that temporal dependencies are preserved and used for forecasting.

Normalization of Time Series

Deep learning models are highly sensitive to the scale of input features. Since raw temperature values (e.g., 30°F to 100°F) can lead to unstable gradients and slow convergence, we apply normalization.

We use `MinMaxScaler` from `sklearn.preprocessing` to scale all temperature values to the range $[0, 1]$.

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

where $x_{\min}$ and $x_{\max}$ are computed from the training set only. This prevents information leakage from the validation or test set into the model.

After training and evaluation, predictions are transformed back to the original scale using the inverse transformation:

$$x = x' \cdot (x_{\max} - x_{\min}) + x_{\min}$$

Important Note: The scaler is fit only on the training set, and the same transformation is applied to validation and test sets to preserve temporal integrity.

1. Vanilla Recurrent Neural Network (RNN)

Recurrent Neural Networks (RNNs) are designed to handle sequential data by maintaining a hidden state that captures information from previous time steps. The core idea is to feed the hidden state from the previous time step back into the model, allowing it to ”remember” previous information.

Internal Architecture

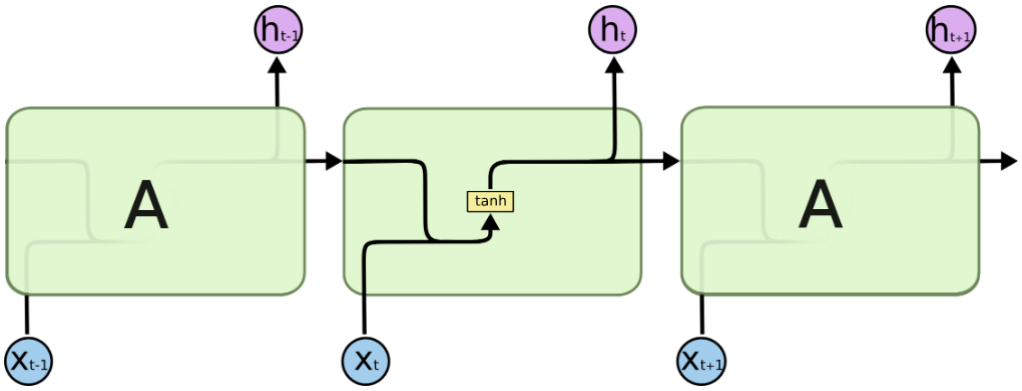

Figure 9: LSTM model prediction vs actual temperature on test data (one-step forecast)

The RNN updates its hidden state h_t at each time step t using the current input x_t and the previous hidden state h_{t-1}:

$$h_t = \tanh(W_h x_t + U_h h_{t-1} + b_h)$$

where:

- x_t is the input at time t

- h_{t-1} is the hidden state from the previous time step
- W_h, U_h, and b_h are learnable weights and bias
- tanh is the hyperbolic tangent activation function

At the final time step of the input sequence (i.e., the last day in the 30-day window), the hidden state h_{30} is passed through a fully connected dense layer to generate the predicted temperature for the next day:

$$\hat{y} = W_o h_{30} + b_o$$

Results and Analysis

Figure shows the predicted vs actual `TMAX` values on the test set using the RNN model.

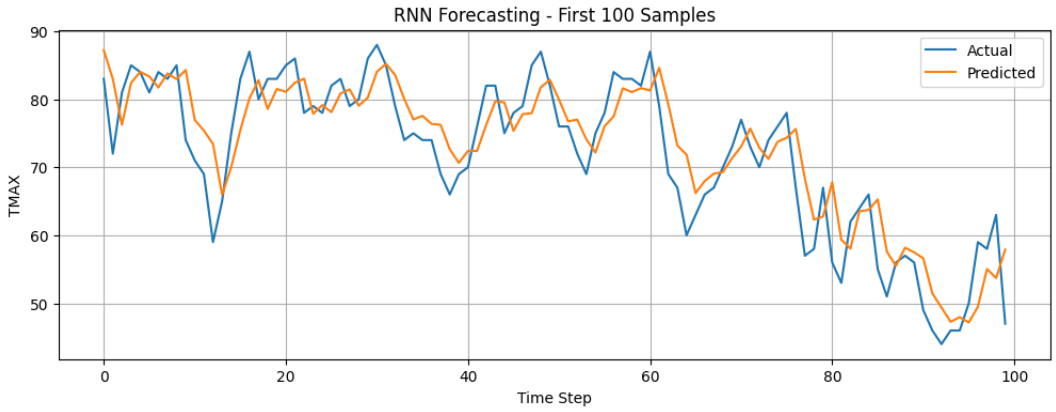

Figure 10: RNN model prediction vs actual temperature on test data (one-step forecast)

From the plot, it is evident that the RNN model captures the previos trend with small variations. Now, since daily temperatures don't change dramatically, and because our target is just one day ahead, the model learns to copy or slightly adjust the last few values of the input.

Hence, the prediction looks like a delayed version of the real data. Why This Happens: Short Horizon: Predicting only 1 step ahead means the input and target are highly correlated.

Stationarity: Temperature is often locally smooth and autocorrelated.

Model Bias: With minimal transformations, the model tends to act like a glorified average or "smart shifter".

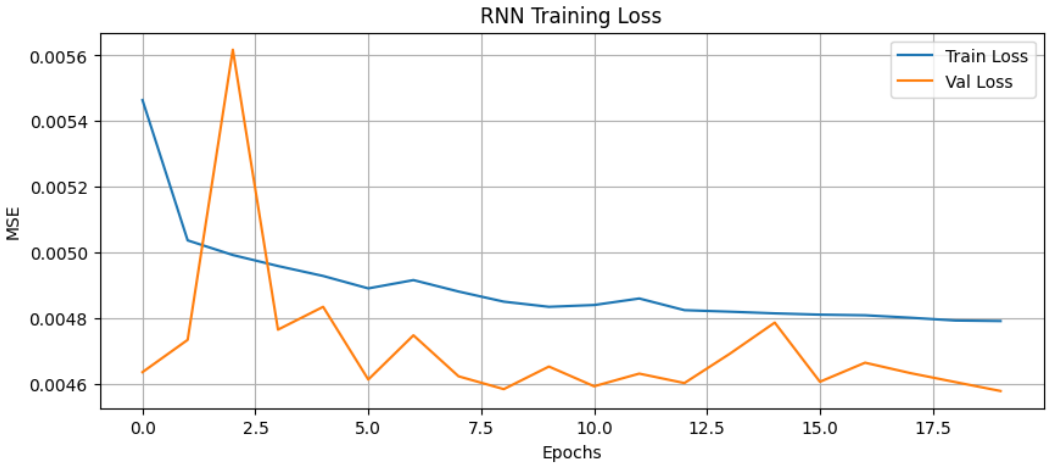

Figure 11: Training loss curve for RNN over 20 epochs

Hyperparameter Tuning for Vanilla RNN

To improve the forecasting performance of the vanilla RNN in the one-step univariate setting, we performed manual grid search over a range of hyperparameters. The objective was to identify the best combination that minimizes the validation RMSE.

Hyperparameter Search Space

The following hyperparameters were tuned:

- **Number of RNN Units**: {32, 64, 128}
- **Dropout Rate**: {0.0, 0.2}
- **Learning Rate (Adam optimizer)**: {0.001, 0.0005}
- **Batch Size**: {32, 64}
- **Epochs**: 30 (fixed for comparability)

Evaluation Metric

The primary evaluation metric for model selection was the Root Mean Squared Error (RMSE) on the validation set. Each configuration was trained using the training set and evaluated on a held-out validation set (chronologically split from the time series data).

Table 2: vanilla RNN Model Hyperparameters

Parameter	**Value**
Window Size	30
RNN Units	64
Dropout Rate	0.2
Learning Rate	0.001
Batch Size	32
Epochs	30
Optimizer	Adam
Loss Function	Mean Squared Error

This configuration yielded the lowest validation RMSE, indicating superior generalization ability on unseen data.

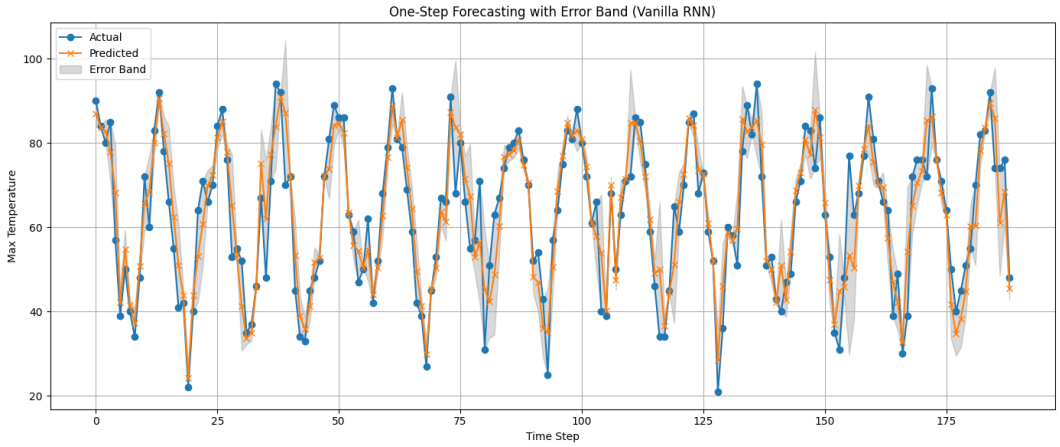

Figure 12: Forecasting using RNN tunned with optimal parameters

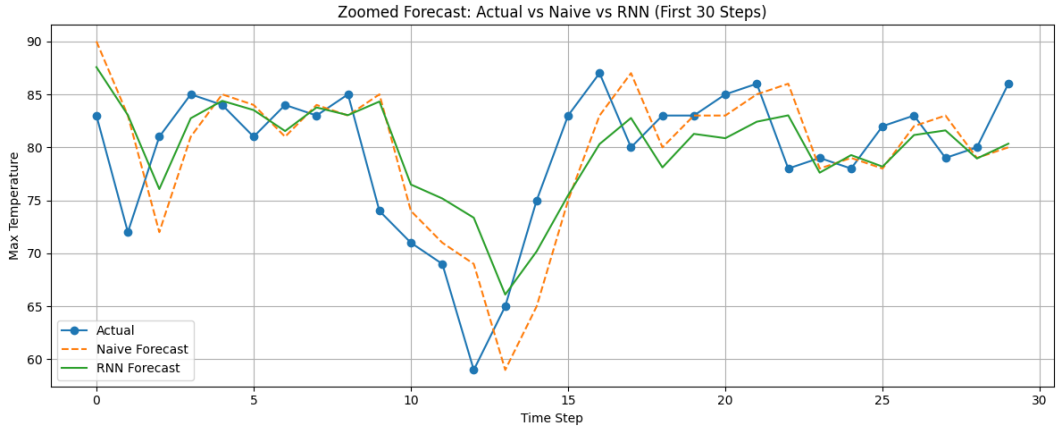

Figure 13: Forecasting using RNN tunned with optimal parameters for 30 days

Even after finding the best hyper-parameters we have the same issue as previous. So we now move to new data preparation strategy.

Data Preparation Using Non-Overlapping Windows for Time Series Forecasting

Given a univariate time series

$$\{x_t\}_{t=1}^{N}, \quad \text{where } x_t \in R$$

we aim to prepare the data for training a recurrent neural network (RNN) using non-overlapping sliding windows of fixed length $w \in N$.

1. Normalization

Before sequence generation, we normalize the input time series using Min-Max scaling:

$$x_t^{(\text{scaled})} = \frac{x_t - \min(x)}{\max(x) - \min(x)}$$

where $\min(x)$ and $\max(x)$ are the minimum and maximum values of the time series.

2. Sequence Construction

To construct the input-output pairs $(\mathbf{X}_i, y_i)$ for the model, we divide the time series into non-overlapping windows of size w. Each input $\mathbf{X}_i \in R^w$ consists of a block of w consecutive time steps, and the target $y_i \in R$ is the immediate next value after the window:

$$\mathbf{X}_i = [x_{iw+1}, x_{iw+2}, \ldots, x_{iw+w}] \quad \text{and} \quad y_i = x_{iw+w+1}$$

for $i = 0, 1, 2, \ldots, \left\lfloor \frac{N-w-1}{w} \right\rfloor$.

This creates:

$$n_{\text{samples}} = \left\lfloor \frac{N - w - 1}{w} \right\rfloor$$

training examples from the full dataset of N points.

3. Data Reshaping for RNN

Each input sequence $\mathbf{X}_i \in R^w$ is reshaped into 3D format for RNNs:

$$\mathbf{X}_i^{(\text{RNN})} \in R^{w \times 1}$$

to represent w time steps with 1 feature per step.

4. Train/Validation/Test Split

Let the total number of constructed sequences be $n = \text{len}(X)$. We divide them as follows:

- Training set: $70\% \Rightarrow n_{\text{train}} = \lfloor 0.7 \cdot n \rfloor$
- Validation set: $20\% \Rightarrow n_{\text{val}} = \lfloor 0.2 \cdot n \rfloor$
- Test set: Remaining samples

Thus,

$$\begin{aligned} \mathbf{X}_{\text{train}} &= \{\mathbf{X}_i\}_{i=1}^{n_{\text{train}}}, & y_{\text{train}} &= \{y_i\}_{i=1}^{n_{\text{train}}} \\ \mathbf{X}_{\text{val}} &= \{\mathbf{X}_i\}_{i=n_{\text{train}}+1}^{n_{\text{train}}+n_{\text{val}}}, & y_{\text{val}} &= \{y_i\}_{i=n_{\text{train}}+1}^{n_{\text{train}}+n_{\text{val}}} \\ \mathbf{X}_{\text{test}} &= \{\mathbf{X}_i\}_{i=n_{\text{train}}+n_{\text{val}}+1}^{n}, & y_{\text{test}} &= \{y_i\}_{i=n_{\text{train}}+n_{\text{val}}+1}^{n} \end{aligned}$$

This partitioning ensures that training, validation, and test sets do not overlap in time, which is essential for proper temporal forecasting tasks.

Advantages

- **No Data Overlap:** Each window is independent and distinct.
- **Generalization:** Helps to avoid overfitting by ensuring the model learns from diverse data segments.
- **Improved Robustness:** Training on non-overlapping windows can help improve the model's ability to generalize across different time periods.

Table 3: vanilla RNN Model Hyperparameters with non overlapping data

Parameter	Value
Window Size	30
RNN Units	64
Dropout Rate	0.2
Learning Rate	0.001
Batch Size	64
Epochs	30
Optimizer	Adam
Loss Function	Mean Squared Error

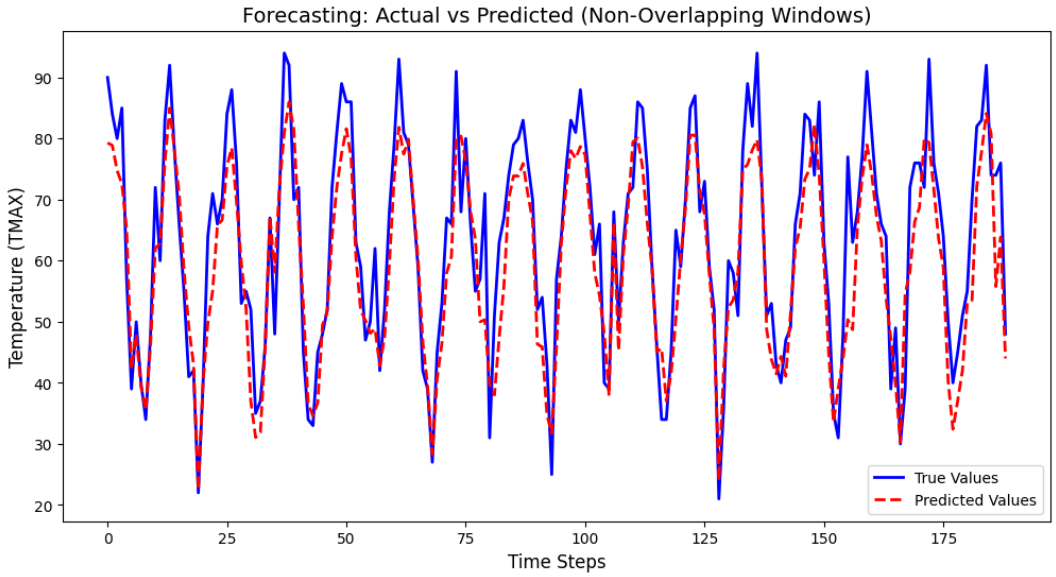

Figure 14: Forecasting using RNN with no overlapping data

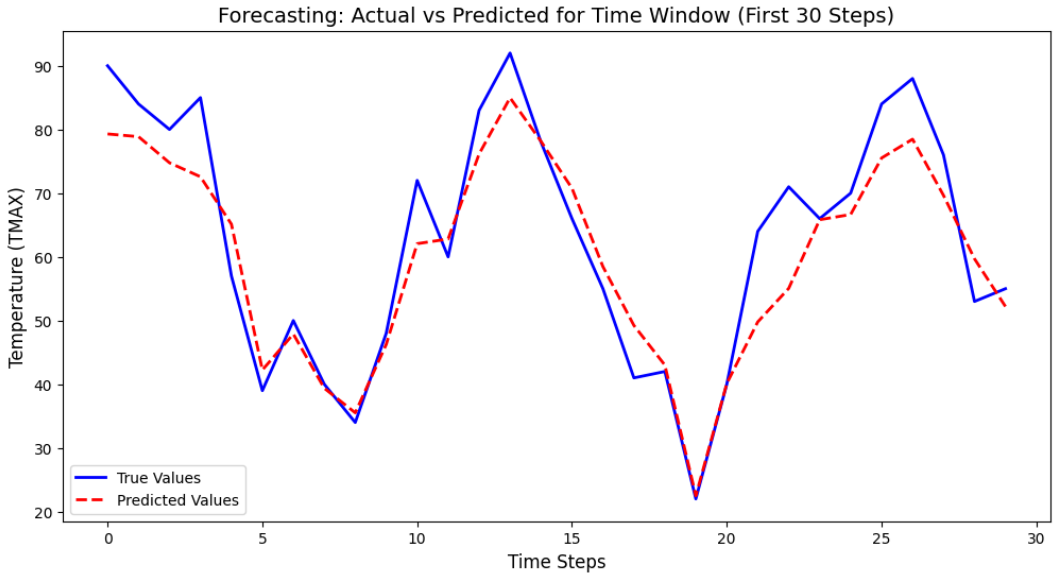

Figure 15: Forecasting using RNN with no overlapping data for 30 days

Table 4: Evaluation Metrics for Best RNN Model

Metric	**Value**
Mean Squared Error (MSE)	60.7405
Root Mean Squared Error (RMSE)	7.7936
Mean Absolute Error (MAE)	6.3596
R-squared (R^2)	0.8043

Hyper-parameter tuning

Table 5: vanilla RNN Model Best Hyperparameters with non overlapping data

Parameter	**Value**
Window Size	30
RNN Units	32
Dropout Rate	0.2
Learning Rate	0.001
Batch Size	32
Epochs	30
Optimizer	Adam
Loss Function	Mean Squared Error

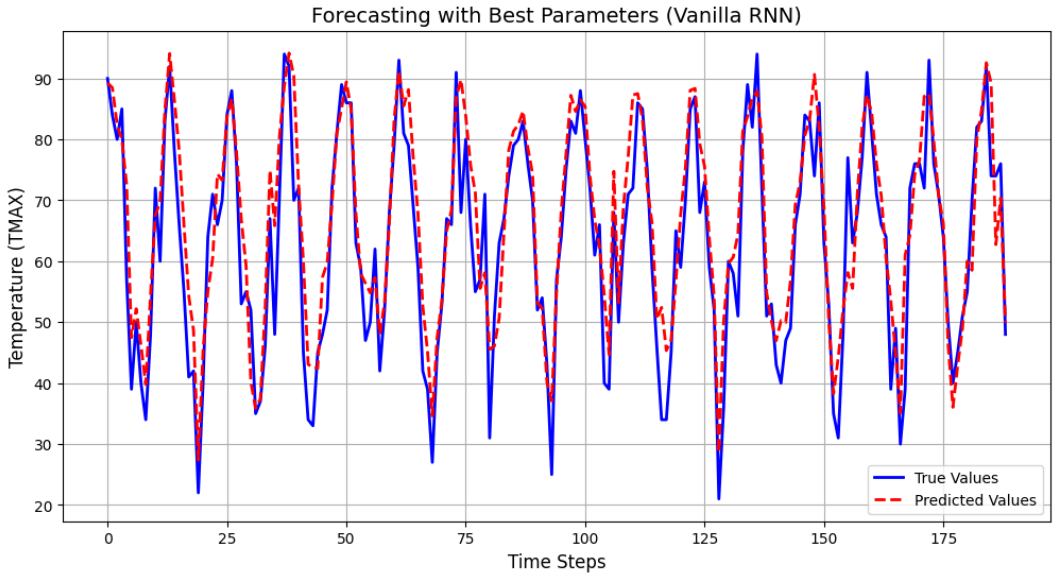

Figure 16: Forecasting using RNN with no overlapping data

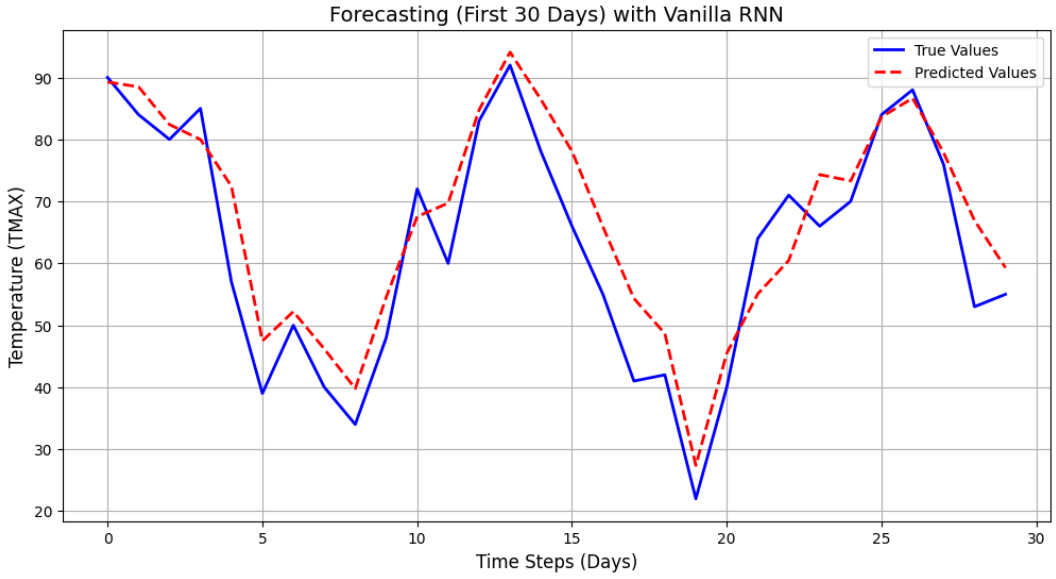

Figure 17: Forecasting using RNN with no overlapping data for 30 days

Table 6: Evaluation Metrics for Best RNN Model

Metric	**Value**
Mean Squared Error (MSE)	44.8287
Root Mean Squared Error (RMSE)	6.6954
Mean Absolute Error (MAE)	5.2055
R-squared (R^2)	0.8555

2. Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks were introduced to mitigate the limitations of vanilla RNNs, especially their inability to capture long-range dependencies due to vanishing gradients. LSTMs achieve this by incorporating gating mechanisms that regulate the flow of information through the network.

Internal Architecture

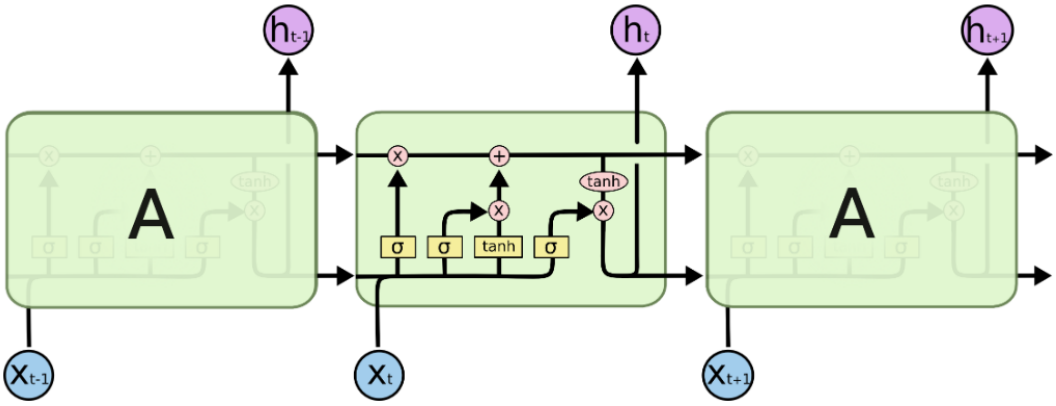

Figure 18: LSTM model prediction vs actual temperature on test data (one-step forecast)

LSTM cells maintain two internal states: the cell state c_t and the hidden state h_t. The computation involves several gates:

$$
\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) && \text{(Forget gate)} \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) && \text{(Input gate)} \\
\tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) && \text{(Candidate cell state)} \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t && \text{(Updated cell state)} \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) && \text{(Output gate)} \\
h_t &= o_t \odot \tanh(c_t) && \text{(Hidden state)}
\end{aligned}
$$

Here, σ is the sigmoid function and $\odot$ denotes element-wise multiplication.

Results and Analysis

The performance of the LSTM model on the test set is shown in Figure 22.

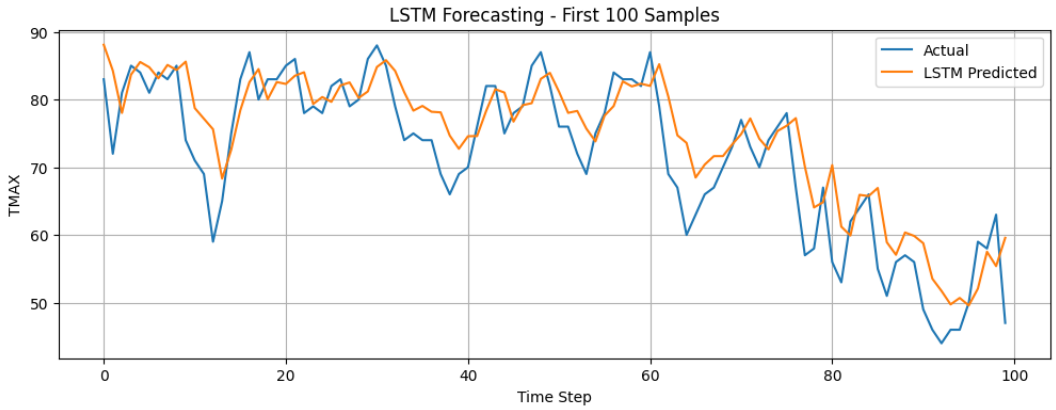

Figure 19: LSTM model prediction vs actual temperature on test data (one-step forecast)

The same issue that we have seen in vanilla RNN also happens with LSTM. So we now directly consider non overlapping windows only.

Hyperparameter Grid Search

Hyperparameters to Optimize

The hyperparameters that are optimized in this approach include:

- **Units**: The number of units in the LSTM layer. This controls the model's capacity to capture temporal patterns.
- **Dropout Rate**: A regularization method to prevent overfitting by randomly setting a fraction of the input units to zero during training.
- **Learning Rate**: Controls the step size at each iteration while updating model weights.
- **Batch Size**: Defines how many training samples are used in one forward/backward pass.
- **Epochs**: The number of times the model is trained over the entire dataset.

The grid search will evaluate each combination of these hyperparameters. We define the grid as:

$$\text{Grid} = \{(u, d, l, b, e) \mid u \in \text{Units}, d \in \text{Dropout}, l \in \text{Learning Rate}, b \in \text{Batch Size}, e \in \text{Epochs}\}$$

where:

$$\text{Units} = \{32, 64, 128\}, \quad \text{Dropout} = \{0.0, 0.2\}, \quad \text{Learning Rate} = \{0.001, 0.0005\}, \quad \text{Batch Size} = \{32, 64\}, \quad \text{Epoc}$$

Grid Search Methodology

A grid search is performed where each combination of hyperparameters is evaluated. For each set of parameters, a model is trained, and its performance is evaluated based on the validation RMSE.

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

The hyperparameter combination that yields the lowest RMSE on the validation set is selected as the best configuration.

Model Construction and Training

Model Architecture

The model consists of the following layers:

- One LSTM layer with the specified number of units.
- A Dropout layer with the specified dropout rate.
- A Dense output layer with one unit for regression.

The model is compiled using the Adam optimizer with the chosen learning rate and Mean Squared Error (MSE) loss function:

$$\mathcal{L}(\hat{y}, y) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Training the Model

For each combination of hyperparameters, the model is trained using the following update rule:

$$w_{t+1} = w_t - \eta \nabla_w \mathcal{L}$$

where: - w_t represents the model weights at time step t, - η is the learning rate, - $\nabla_w \mathcal{L}$ is the gradient of the loss function with respect to the model weights.

The model is trained using the specified batch size and the number of epochs.

Model Evaluation

Performance Metric

The performance of each trained model is evaluated using the Root Mean Squared Error (RMSE) on the validation set:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

where $\hat{y}_i$ are the predicted values, and y_i are the actual values.

Selection of Best Hyperparameters

Best Hyperparameter Combination

Once all models are trained and evaluated, the hyperparameter combination that results in the lowest RMSE on the validation set is selected as the best configuration. These best hyperparameters are then used to train the final model on the entire training dataset.

Final Model Training

The final model is trained using the best hyperparameters on the full training dataset. After training, the model is evaluated on the test set to estimate its generalization ability.

tableAll LSTM Hyperparameter Tuning Results

Units	Dropout	Learning Rate	Batch Size	Epochs	Val RMSE
32	0.0	0.0010	32	30	0.073604
128	0.0	0.0010	32	30	0.073617
64	0.0	0.0010	32	30	0.074216
64	0.2	0.0010	32	30	0.075906
128	0.2	0.0010	32	30	0.076018
32	0.0	0.0010	64	30	0.077147
128	0.0	0.0005	32	30	0.077432
32	0.2	0.0010	32	30	0.077608
128	0.2	0.0005	32	30	0.078088
64	0.2	0.0005	32	30	0.078094
128	0.2	0.0010	64	30	0.078128
32	0.2	0.0005	32	30	0.080035
32	0.0	0.0005	32	30	0.080319
128	0.0	0.0005	64	30	0.080856
64	0.0	0.0010	64	30	0.080973
128	0.0	0.0010	64	30	0.081131
32	0.2	0.0010	64	30	0.081216
64	0.2	0.0005	64	30	0.081278
32	0.0	0.0005	64	30	0.081382
128	0.2	0.0005	64	30	0.082207
64	0.0	0.0005	64	30	0.083640
64	0.2	0.0010	64	30	0.083923
32	0.2	0.0005	64	30	0.085240
64	0.0	0.0005	32	30	0.089889

tableFinal Validation Loss

Index	Final Val Loss
0	0.005418
1	0.005419
2	0.005508
3	0.005762
4	0.005779
5	0.005952
6	0.005996
7	0.006023
8	0.006098
9	0.006099
10	0.006104
11	0.006406
12	0.006451
13	0.006538
14	0.006557
15	0.006582
16	0.006596
17	0.006606
18	0.006623
19	0.006758
20	0.006996
21	0.007043
22	0.007266
23	0.008080

Results Visualization

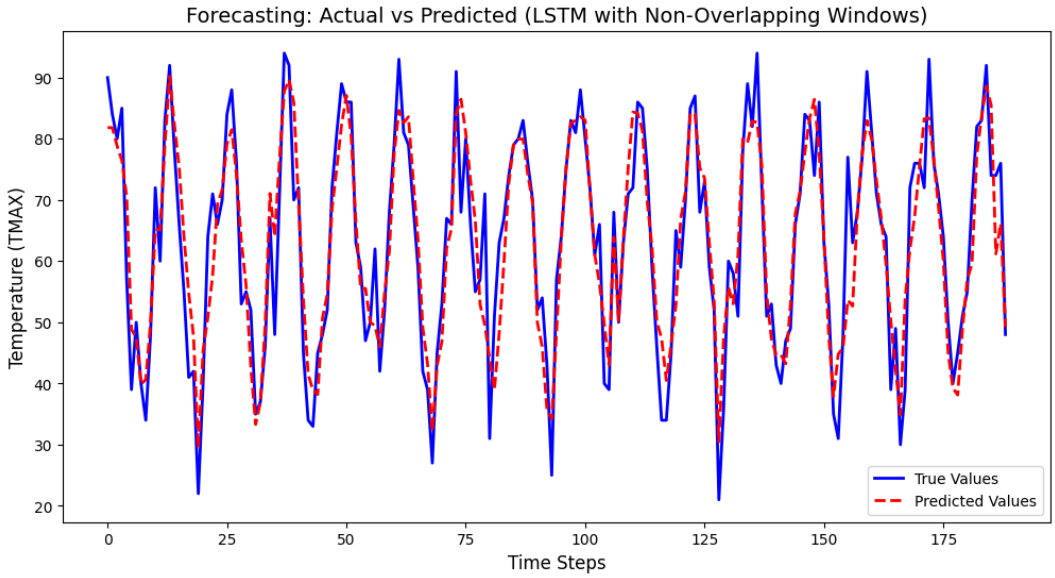

Figure 20: LSTM model prediction vs actual temperature on test data (one-step forecast)

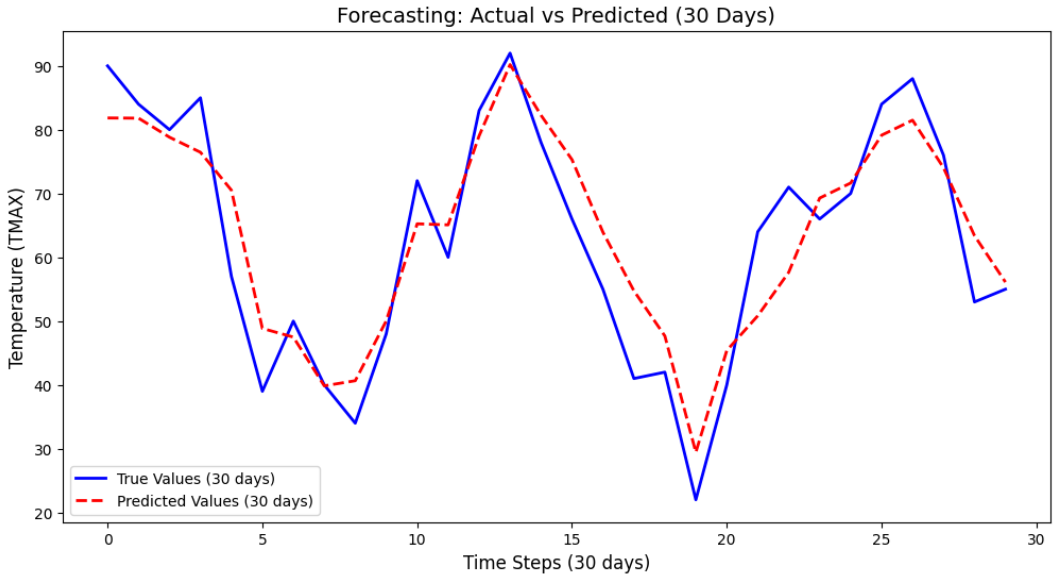

Figure 21: LSTM model prediction vs actual temperature on test data (one-step forecast)

Table 7: Evaluation Metrics on Entire Test Set

Metric	**Value**
Mean Absolute Error (MAE)	5.5505
Mean Squared Error (MSE)	50.8684
Root Mean Squared Error (RMSE)	7.1322
R-squared (R^2)	0.8361

3. Gated Recurrent Unit (GRU)

Gated Recurrent Units (GRUs) are a streamlined version of LSTMs introduced by Cho et al. (2014). GRUs simplify the internal architecture by combining the forget and input gates into a single *update gate*, and merging the cell and hidden states. This simplification reduces computational complexity while preserving the ability to model long-term dependencies.

Internal Architecture

The GRU cell consists of the following components:

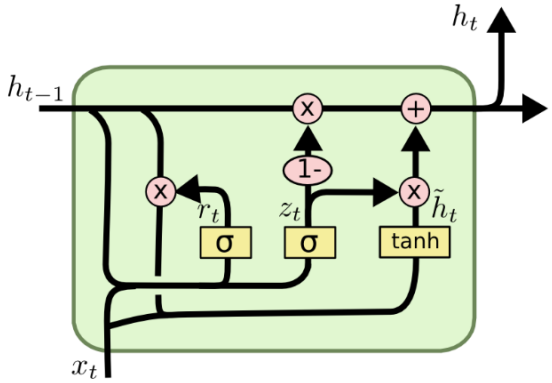

Figure 22:

$$
\begin{aligned}
z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) && \text{(Update gate)} \\
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) && \text{(Reset gate)} \\
\tilde{h}_t &= \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) && \text{(Candidate activation)} \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t && \text{(Final hidden state)}
\end{aligned}
$$

Compared to LSTM, GRU has fewer parameters and is often faster to train while achieving similar or slightly better performance on some tasks.

Results and Analysis

The GRU model's one-step forecasting performance on the test set is visualized in Figure 27.

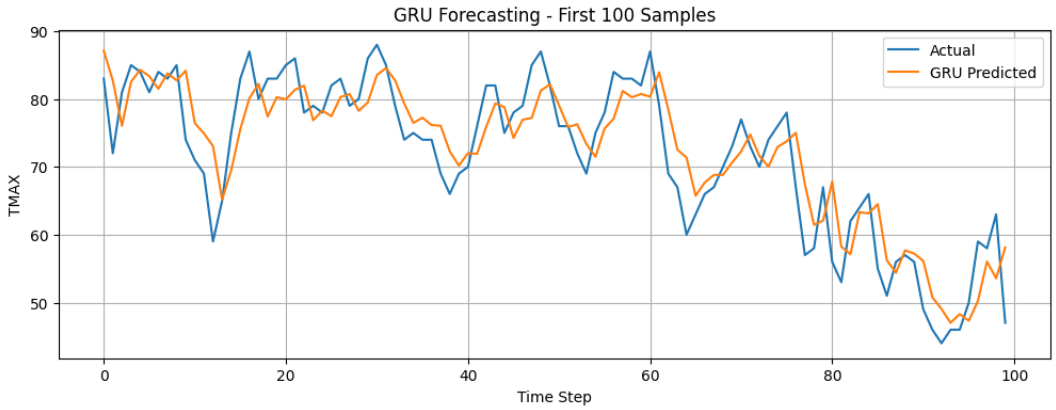

Figure 23: GRU model prediction vs actual temperature on test data (one-step forecast)

the same issue of highly overlapped data is happeing with GRU as well so we choose non overlapping data.

Table 8: vanilla RNN Model Hyperparameters with non overlapping data

Parameter	**Value**
Window Size	30
Units	32
Dropout Rate	0.2
Learning Rate	0.001
Batch Size	32
Epochs	30
Optimizer	Adam
Loss Function	Mean Squared Error

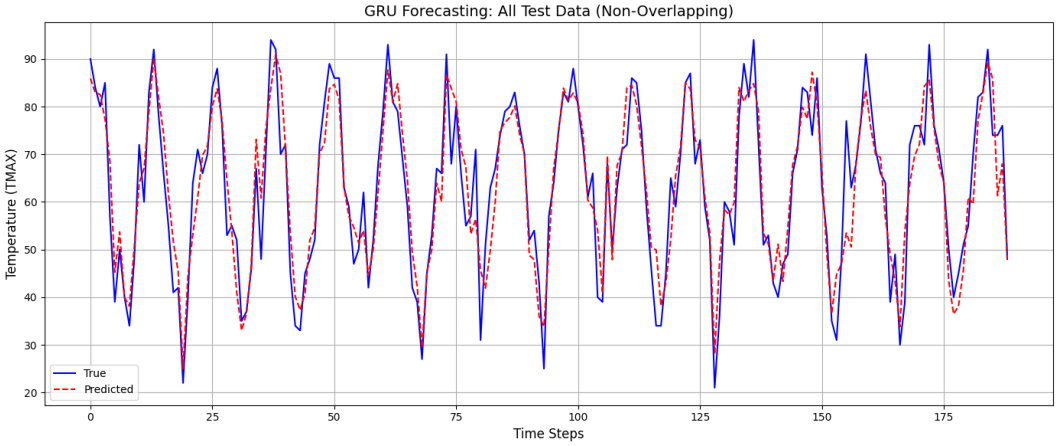

Figure 24: GRU model prediction vs actual temperature on test data (one-step forecast)

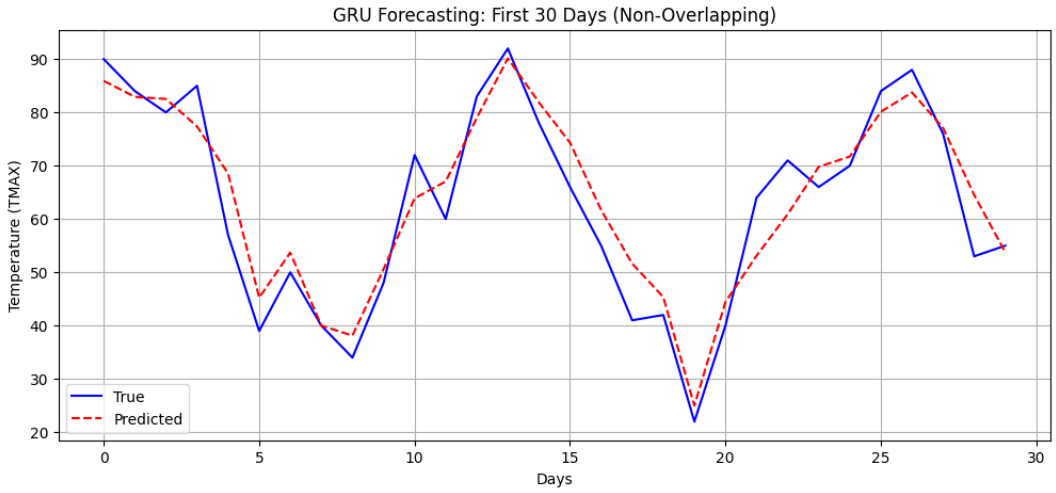

Figure 25: GRU model prediction vs actual temperature on test data (one-step forecast)

Table 9: Evaluation Metrics on Test Set (Latest)

Metric	**Value**
Mean Squared Error (MSE)	48.6409
Root Mean Squared Error (RMSE)	6.9743
Mean Absolute Error (MAE)	5.4318
R-squared (R^2)	0.8433

Hyper-parameter Tuning Results

Table 10: All GRU Hyperparameter Tuning Results

Units	Dropout	LR	Batch	Epochs	Val RMSE
64	0.0	0.0010	32	30	0.070572
64	0.2	0.0010	32	30	0.070608
32	0.0	0.0010	64	30	0.070806
32	0.2	0.0010	32	30	0.071166
64	0.0	0.0005	32	30	0.071975
64	0.2	0.0005	32	30	0.072287
32	0.0	0.0010	32	30	0.072693
32	0.0	0.0005	64	30	0.072734
64	0.0	0.0010	64	30	0.073061
32	0.2	0.0005	32	30	0.073467
32	0.0	0.0005	32	30	0.073877
64	0.0	0.0005	64	30	0.073895
64	0.2	0.0010	64	30	0.073942
32	0.2	0.0005	64	30	0.074279
32	0.2	0.0010	64	30	0.074860
64	0.2	0.0005	64	30	0.076194

Table 11: Final Validation Loss (GRU)

Final Val Loss
0.004980
0.004985
0.005013
0.005065
0.005180
0.005225
0.005284
0.005290
0.005338
0.005397
0.005458
0.005460
0.005467
0.005517
0.005604
0.005805

1 Forecasting using optimal parameters

Table 12: vanilla RNN Model Hyperparameters with non overlapping data

Parameter	Value
Window Size	30
Units	32
Dropout Rate	0.2
Learning Rate	0.001
Batch Size	32
Epochs	30
Optimizer	Adam
Loss Function	Mean Squared Error

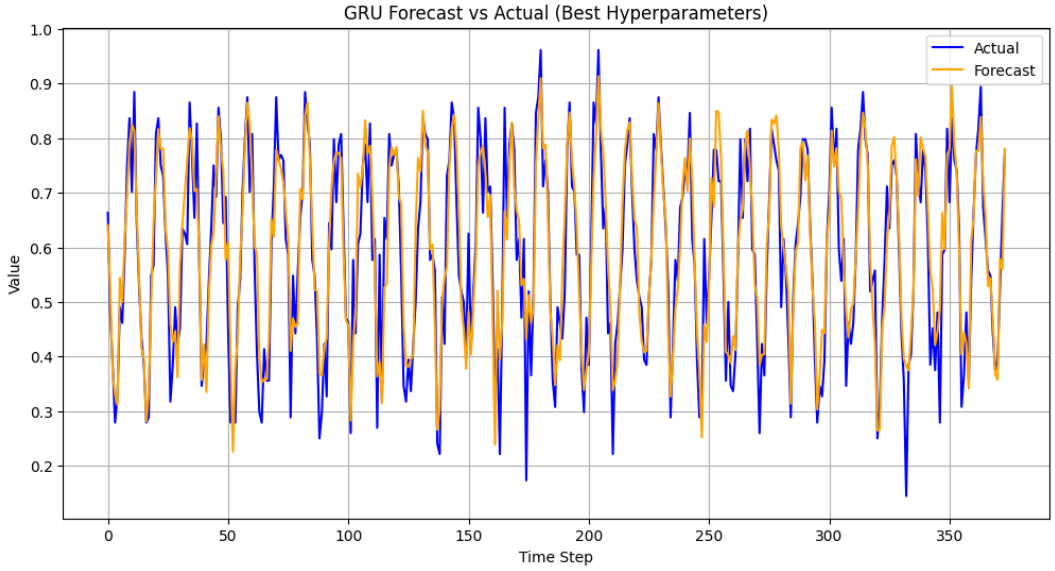

Figure 26: GRU model prediction vs actual temperature on test data (one-step forecast)

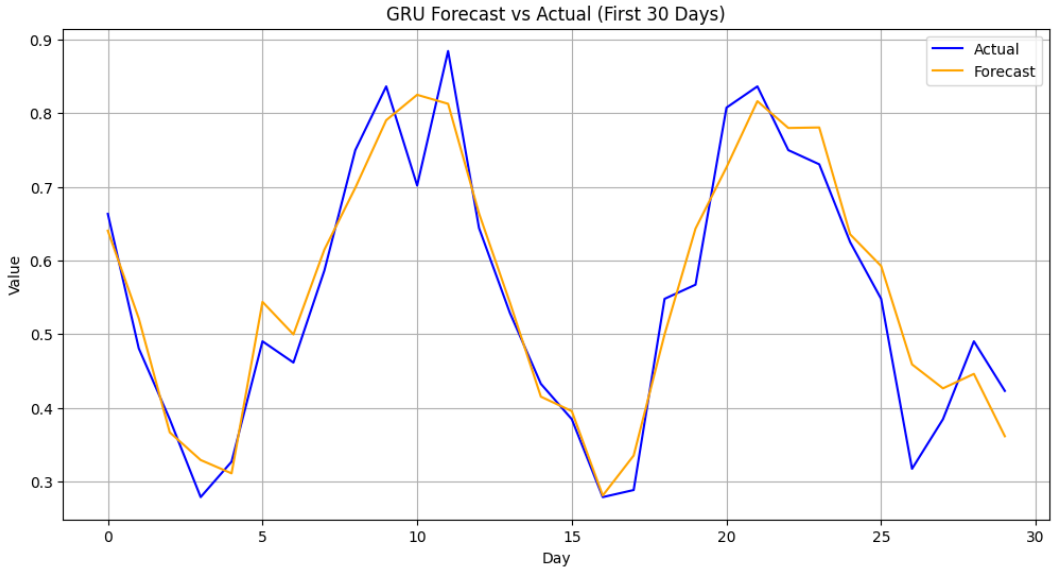

Figure 27: GRU model prediction vs actual temperature on test data (one-step forecast)

Table 13: Evaluation Metrics on Test Set (Updated)

Metric	**Value**
Mean Squared Error (MSE)	44.8287
Root Mean Squared Error (RMSE)	6.6954
Mean Absolute Error (MAE)	5.2055
R-squared (R^2)	0.8555