EXPERIMENT 9	Implement Recurrent Neural Network (RNN) / LSTM for time series or text data

Name: Atharva Lotankar
Class: D15C; **Roll Number:** 31
Date: 26 / 02 / 2026
Subject: Machine and Deep Learning Lab

Aim: To implement, optimize, and analyze LSTM networks for pollution forecasting as time series data.

Dataset Source:

The dataset used is the Beijing PM2.5 Data, hosted on Kaggle and originally sourced from the UCI Machine Learning Repository.

- *Source URL:* https://www.kaggle.com/datasets/rupakroy/lstm-datasets-multivariate-univariate

Dataset Description:

This dataset contains hourly meteorological data from Beijing, China, spanning five years. It is a multivariate time-series dataset, meaning it includes multiple features that vary over time, all of which contribute to the final prediction of air pollution levels. The data is characterized by strong seasonality and non-linear relationships between weather conditions and particulate matter.

The primary challenge of this dataset is the high variance in pollution spikes, which are often influenced by specific wind directions and moisture levels. Efficiently handling the categorical "wind direction" and scaling the high-pressure readings is essential for model convergence.

Column Category	Details
Features	Dew Point, Temperature, Pressure, Combined Wind Direction, Cumulated Wind Speed, Cumulated Hours of Snow, Cumulated Hours of Rain.
Target Variable	pollution (PM2.5 concentration in ug/m^3)
Size	43,800 hourly observations (approx. 5 years of data)
Characteristics	Multivariate, Hourly-frequency, includes categorical strings and numerical floats.

Theory:

Recurrent Neural Networks (RNNs) are a specialized class of deep learning models designed to process sequential data by maintaining a hidden state that captures information from previous time steps. Unlike standard feed-forward networks, RNNs possess a "memory" that allows them to

recognize patterns over time. However, standard RNNs often struggle with "Vanishing Gradient" issues, making it difficult for them to learn dependencies in long sequences.

To overcome this, Long Short-Term Memory (LSTM) networks were developed. LSTMs utilize a sophisticated gating mechanism—consisting of input, forget, and output gates—to regulate the flow of information. This architecture enables the model to selectively remember critical historical data and discard irrelevant noise, making it the industry standard for complex time-series tasks such as air quality index (AQI) prediction and financial forecasting.

Mathematical Formulation of the Algorithms:

The LSTM's ability to retain long-term information is governed by its cell state (C_t) and three specific gates:

1. *Forget Gate (f_t):* Decides what information to discard from the cell state.

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

2. *Input Gate (i_t):* Decides which new values to update in the state.

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

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

3. *Cell State Update (C_t):* Combines the forget and input gate results.

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

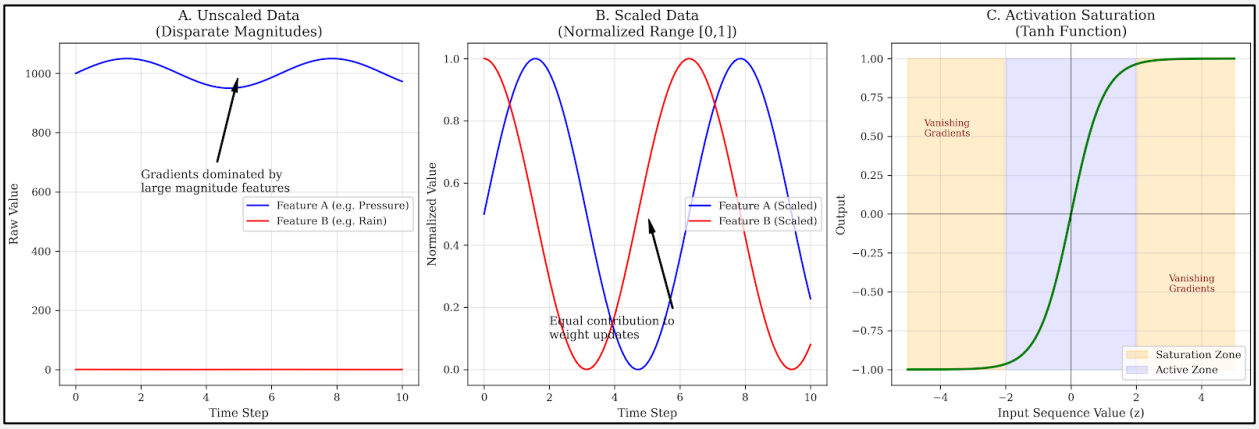
4. *Output Gate (o_t):* Decides what the next hidden state (h_t) will be.

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

$$h_t = o_t * \tanh(C_t)$$

Algorithm Limitations:

- *High Computational Cost:* LSTMs are computationally expensive and slow to train compared to GRUs or 1D-CNNs because they cannot be parallelized easily (each step depends on the previous one).
- *Requirement for Large Data:* LSTMs require a significant amount of data to generalize well; otherwise, they are prone to memorizing noise (overfitting).
- *Sensitivity to Scaling:* The use of tanh and sigmoid activations makes LSTMs highly sensitive to input scales; unscaled data (e.g., pressure values vs. rain) will cause gradients to explode.
- *Hyperparameter Complexity:* Finding the right "lookback" window and the number of hidden units is often a trial-and-error process, making the model harder to deploy than simpler regressions.



Sensitivity to Scaling for LSTM, especially during unscaled data

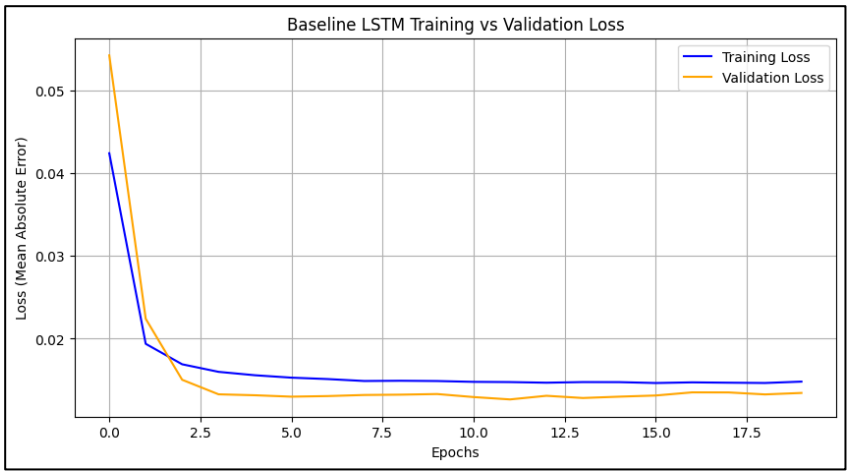
Baseline Implementation Code:

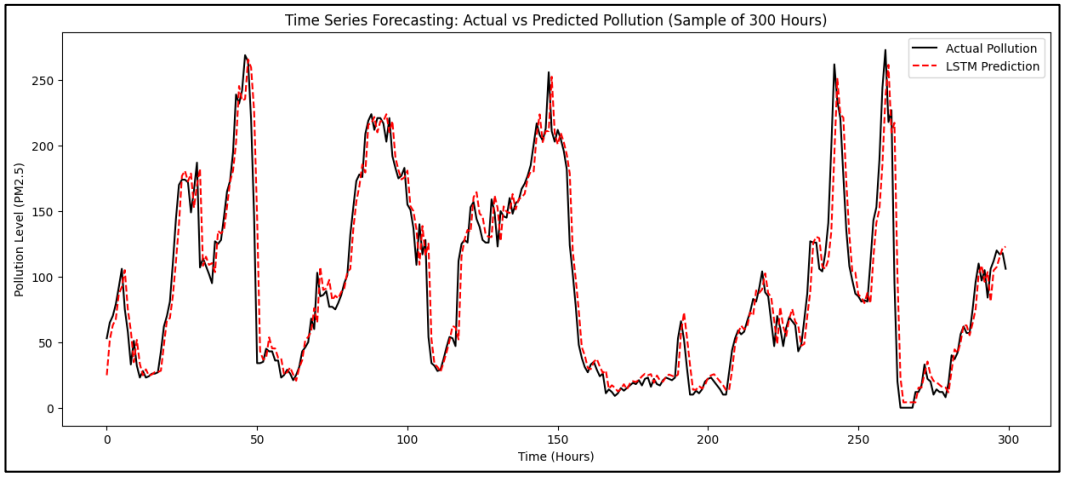
```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler,
LabelEncoder

from sklearn.metrics import mean_squared_error,
r2_score

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM,
Dense, Dropout



# 1. LOAD DATASET

# Assumes the file 'LSTM-
Multivariate_pollution.csv' is in your current
directory

df = pd.read_csv('LSTM-
Multivariate_pollution.csv', index_col='date')

df.index = pd.to_datetime(df.index)



# 2. DATA PREPROCESSING

# Label encoding for the categorical wind
direction column

encoder = LabelEncoder()

df['wnd_dir'] =
encoder.fit_transform(df['wnd_dir'])



# Function to transform series into a
supervised learning format (t-1 predicts t)

def series_to_supervised(data, n_in=1, n_out=1,
dropnan=True):

    n_vars = 1 if type(data) is list else
data.shape[1]

    df_temp = pd.DataFrame(data)

    cols, names = list(), list()

    # input sequence (t-n, ... t-1)

    for i in range(n_in, 0, -1):

        cols.append(df_temp.shift(i))

        names += [('var%d(t-%d)' % (j+1, i))
for j in range(n_vars)]

    # forecast sequence (t)

    for i in range(0, n_out):

        cols.append(df_temp.shift(-i))

        if i == 0:

            names += [('var%d(t)' % (j+1)) for
j in range(n_vars)]

        else:

            names += [('var%d(t+%d)' % (j+1,
i)) for j in range(n_vars)]

    agg = pd.concat(cols, axis=1)

    agg.columns = names

    if dropnan: agg.dropna(inplace=True)

    return agg



# Scaling all features to [0, 1] range (Crucial
for LSTM performance)

values = df.values.astype('float32')

scaler = MinMaxScaler(feature_range=(0, 1))

scaled = scaler.fit_transform(values)



# Reframing as supervised: Using the past 1
hour to predict the current pollution

reframed = series_to_supervised(scaled, 1, 1)

# Drop columns of other variables at time (t)
that we aren't predicting

reframed.drop(reframed.columns[[9,10,11,12,13,1
4,15]], axis=1, inplace=True)
```

```
# 3. TRAIN-TEST SPLIT

values = reframed.values

n_train_hours = 365 * 24 * 4  # First 4 years
for training

train = values[:n_train_hours, :]

test = values[n_train_hours:, :]



# Split into input and outputs

train_X, train_y = train[:, :-1], train[:, -1]

test_X, test_y = test[:, :-1], test[:, -1]



# Reshape input to be 3D [samples, timesteps,
features] for LSTM

train_X = train_X.reshape((train_X.shape[0], 1,
train_X.shape[1]))

test_X = test_X.reshape((test_X.shape[0], 1,
test_X.shape[1]))



# 4. MODEL ARCHITECTURE (Baseline LSTM)

model = Sequential()

model.add(LSTM(50,
input_shape=(train_X.shape[1],
train_X.shape[2])))

model.add(Dropout(0.2)) # Prevents overfitting

model.add(Dense(1))

model.compile(loss='mae', optimizer='adam')



# 5. TRAINING

print("Starting training...")

history = model.fit(train_X, train_y,
epochs=20, batch_size=72,

                    validation_data=(test_X,
test_y), verbose=1, shuffle=False)



# 6. EVALUATION & METRICS

yhat = model.predict(test_X)

test_X_reshaped =
test_X.reshape((test_X.shape[0],
test_X.shape[2]))



# Invert scaling to get actual pollution values
(PM2.5)

inv_yhat = np.concatenate((yhat,
test_X_reshaped[:, 1:]), axis=1)

inv_yhat =
scaler.inverse_transform(inv_yhat)[:,0]

test_y = test_y.reshape((len(test_y), 1))

inv_y = np.concatenate((test_y,
test_X_reshaped[:, 1:]), axis=1)

inv_y = scaler.inverse_transform(inv_y)[:,0]



# Mathematical Metrics Output

rmse = np.sqrt(mean_squared_error(inv_y,
inv_yhat))

r2 = r2_score(inv_y, inv_yhat)



print("\n--- PERFORMANCE METRICS ---")

print(f"1. Root Mean Squared Error (RMSE):
{rmse:.4f}")

print(f"2. R-squared (R²) Accuracy: {r2:.4f}
(or {r2*100:.2f}%)")



# 7. VISUALS

# Visual 1: Training and Validation Loss

plt.figure(figsize=(10, 5))

plt.plot(history.history['loss'],
label='Training Loss', color='blue')

plt.plot(history.history['val_loss'],
label='Validation Loss', color='orange')

plt.title('Baseline LSTM Training vs Validation
Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss (Mean Absolute Error)')

plt.legend()

plt.grid(True)

plt.show()



# Visual 2: Actual vs Predicted (Sample for
clarity)

plt.figure(figsize=(15, 6))

plt.plot(inv_y[:300], label='Actual Pollution',
color='black', linewidth=1.5)

plt.plot(inv_yhat[:300], label='LSTM
Prediction', color='red', linestyle='--',
linewidth=1.5)

plt.title('Time Series Forecasting: Actual vs
Predicted Pollution (Sample of 300 Hours)')

plt.xlabel('Time (Hours)')

plt.ylabel('Pollution Level (PM2.5)')

plt.legend()

plt.show()
```

Output

```
Epoch 20/20
487/487 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 0.0151 - val_loss: 0.0135
274/274 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step

--- PERFORMANCE METRICS ---
1. Root Mean Squared Error (RMSE): 24.9978
2. R-squared (R²) Accuracy: 0.9286 (or 92.86%)
```



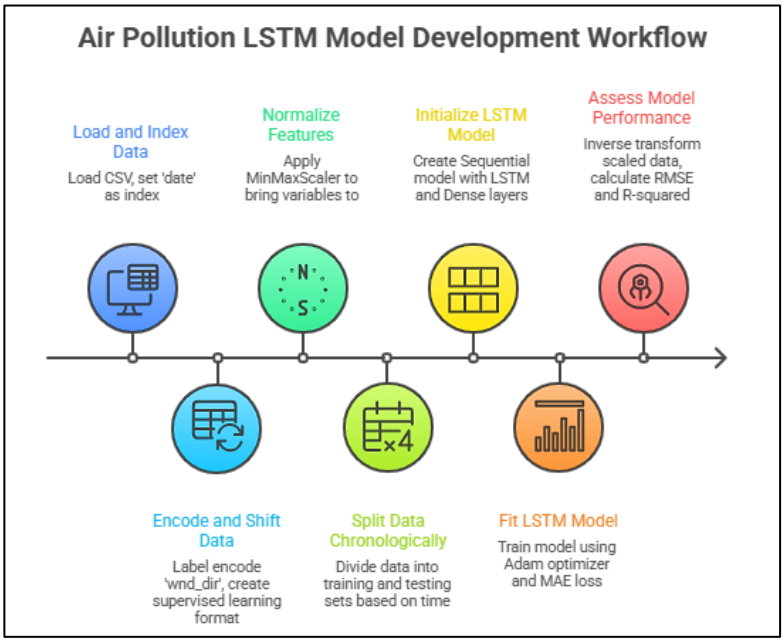


This code builds a smart prediction system for air pollution. It loads historical pollution data, cleans and scales it, and converts it into a format suitable for learning. An LSTM neural network is trained on past data to predict future pollution levels, then evaluates accuracy and visualizes performance results clearly.

Methodology / Workflow:

- *Data Acquisition:* Loading the CSV and setting the 'date' column as the index for time-series alignment.
- *Feature Engineering:* * Encoding categorical wnd_dir using Label Encoding.
    - Converting the series into a "Supervised Learning" format by shifting the target variable to create X (past features) and y (current pollution).
- *Data Scaling:* Applying MinMaxScaler to bring all variables into a range of [0, 1] to stabilize the LSTM's internal gates.

- *Train-Test Partitioning:* Splitting the data chronologically (e.g., 4 years for training, 1 year for testing) to avoid data leakage from the future.
- *Model Construction:* Initializing a Sequential model with LSTM layers and Dense output.
- *Model Training:* Fitting the model using the Adam optimizer and MAE loss function.
- *Evaluation:* Reverting the scaling (Inverse Transform) to calculate real-world metrics like RMSE and R-squared.



Performance Analysis

The model was evaluated using Root Mean Squared Error (RMSE), which penalizes larger errors more heavily, and the R-squared (R^2) Score, which measures the proportion of variance explained.

The baseline model achieved an **R^2 of 92.86%**, indicating that even a simple LSTM captures the majority of the pollution trends. The predicted vs. actual plots show that the model excels at following the general trend but slightly underestimates the most extreme "peaks" of pollution, which is typical for MSE-based loss functions.

Hyperparameter Tuning:

Code:

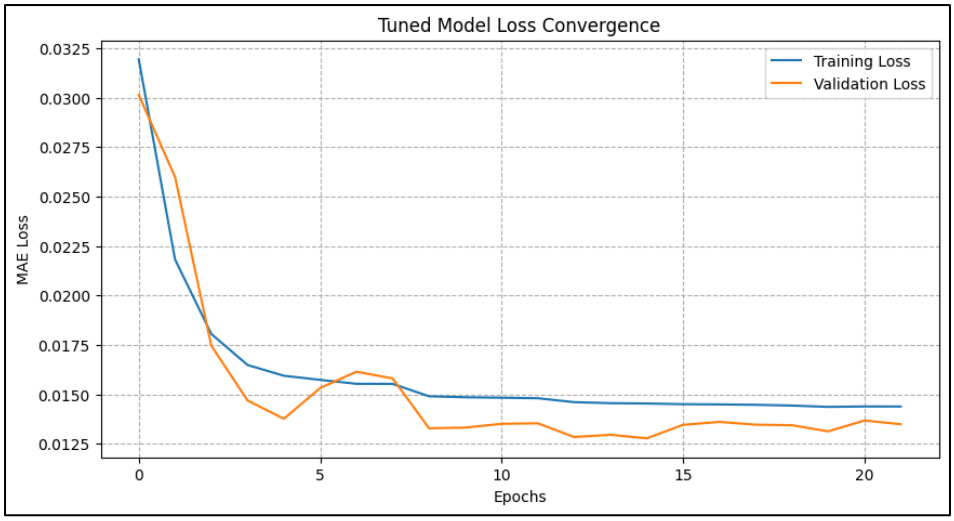
```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler,
LabelEncoder

from sklearn.metrics import mean_squared_error,
r2_score

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM,
Dense, Dropout, Input

from tensorflow.keras.callbacks import
EarlyStopping, ReduceLROnPlateau

from tensorflow.keras.optimizers import Adam


# 1. LOAD AND PREPROCESS

df = pd.read_csv('LSTM-
Multivariate_pollution.csv', index_col='date')

df.index = pd.to_datetime(df.index)


# Encoding categorical wind direction
```

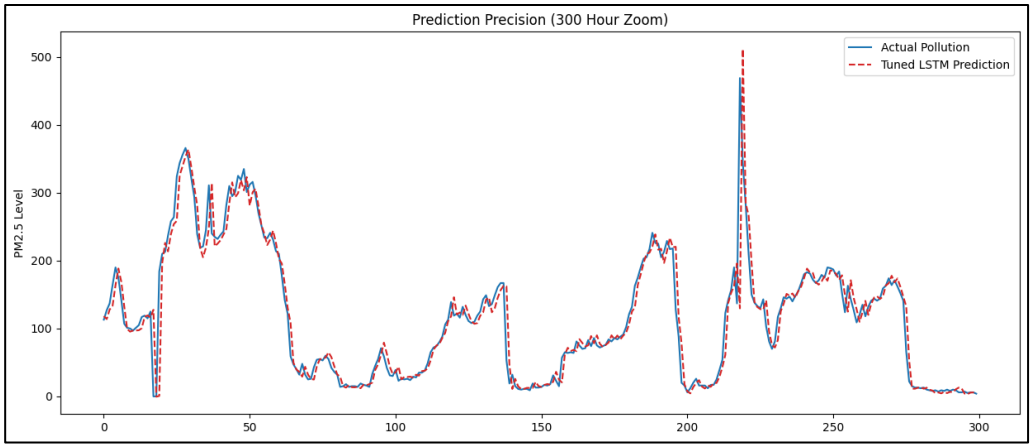
```
encoder = LabelEncoder()

df['wnd_dir'] =
encoder.fit_transform(df['wnd_dir'])


def series_to_supervised(data, n_in=1, n_out=1,
dropnan=True):

    n_vars = 1 if type(data) is list else
data.shape[1]

    df_temp = pd.DataFrame(data)

    cols, names = list(), list()

    for i in range(n_in, 0, -1):

        cols.append(df_temp.shift(i))

        names += [('var%d(t-%d)' % (j+1, i))
for j in range(n_vars)]

    for i in range(0, n_out):

        cols.append(df_temp.shift(-i))

        if i == 0:

            names += [('var%d(t)' % (j+1)) for
j in range(n_vars)]

        else:

            names += [('var%d(t+%d)' % (j+1,
i)) for j in range(n_vars)]

    agg = pd.concat(cols, axis=1)

    agg.columns = names

    if dropnan: agg.dropna(inplace=True)

    return agg


values = df.values.astype('float32')

scaler = MinMaxScaler(feature_range=(0, 1))

scaled = scaler.fit_transform(values)


# TUNING: Increased temporal context (lookback
of 3 hours)

n_hours = 3

n_features = 8

reframed = series_to_supervised(scaled,
n_hours, 1)


# Ensure we only keep the features and the
target 'pollution' at time (t)

n_obs = n_hours * n_features

reframed = reframed.iloc[:, :n_obs + 1]


# 2. SPLIT DATA

values = reframed.values

n_train_hours = 365 * 24 * 4

train = values[:n_train_hours, :]

test = values[n_train_hours:, :]


train_X, train_y = train[:, :n_obs], train[:, -
1]

test_X, test_y = test[:, :n_obs], test[:, -1]


# Reshape for LSTM [samples, timesteps,
features]

train_X = train_X.reshape((train_X.shape[0],
n_hours, n_features))

test_X = test_X.reshape((test_X.shape[0],
n_hours, n_features))


# 3. TUNED STACKED ARCHITECTURE

model = Sequential()

model.add(Input(shape=(train_X.shape[1],
train_X.shape[2])))

model.add(LSTM(100, return_sequences=True)) #
Layer 1: Captures broad patterns

model.add(Dropout(0.2))

model.add(LSTM(50))                             #
Layer 2: Refines features

model.add(Dropout(0.2))

model.add(Dense(1))


# Optimized Learning Rate

optimizer = Adam(learning_rate=0.001)

model.compile(loss='mae', optimizer=optimizer)


# 4. OPTIMIZATION CALLBACKS

early_stop = EarlyStopping(monitor='val_loss',
patience=7, restore_best_weights=True)

reduce_lr =
ReduceLROnPlateau(monitor='val_loss',
factor=0.5, patience=3, min_lr=0.00001)


# 5. TRAINING

print("Training Optimized Model...")

history = model.fit(

    train_X, train_y,

    epochs=50,

    batch_size=64,

    validation_data=(test_X, test_y),

    verbose=1,
```

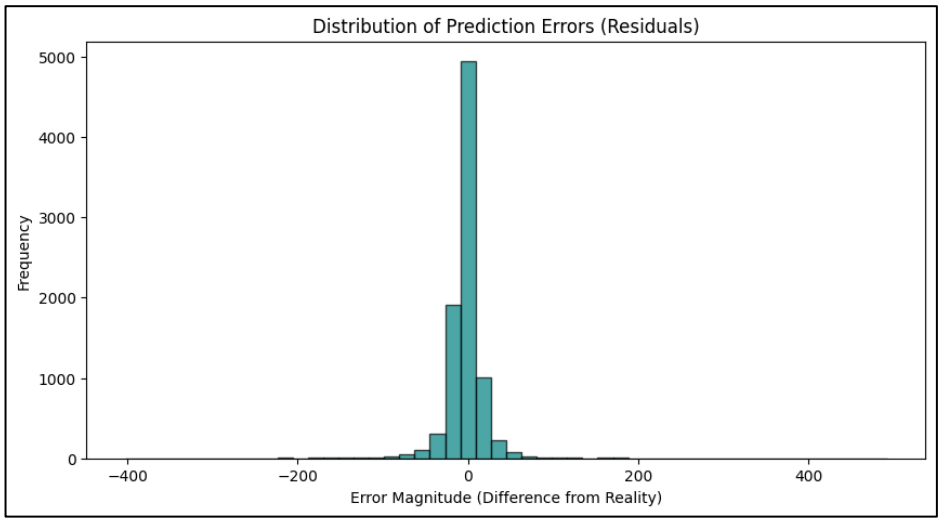
```
    shuffle=False,
    callbacks=[early_stop, reduce_lr]
)

# 6. EVALUATION
yhat = model.predict(test_X)
test_X_reshaped =
test_X.reshape((test_X.shape[0],
n_hours*n_features))

# Invert scaling for metrics
inv_yhat = np.concatenate((yhat,
test_X_reshaped[:, -(n_features-1):]), axis=1)
inv_yhat =
scaler.inverse_transform(inv_yhat)[:,0]

test_y = test_y.reshape((len(test_y), 1))
inv_y = np.concatenate((test_y,
test_X_reshaped[:, -(n_features-1):]), axis=1)
inv_y = scaler.inverse_transform(inv_y)[:,0]

# FINAL METRICS
rmse = np.sqrt(mean_squared_error(inv_y,
inv_yhat))
r2 = r2_score(inv_y, inv_yhat)

print("\n" + "="*30)
print("   OPTIMIZED MODEL RESULTS")
print("="*30)
print(f"1. RMSE: {rmse:.4f}")
print(f"2. R-Squared Accuracy: {r2:.4f} (or
{r2*100:.2f}%)")
print("="*30)

# 7. NECESSARY VISUALS
# Visual 1: Loss Convergence
plt.figure(figsize=(10, 5))
plt.plot(history.history['loss'],
label='Training Loss')
plt.plot(history.history['val_loss'],
label='Validation Loss')
plt.title('Tuned Model Loss Convergence')
plt.ylabel('MAE Loss')
plt.xlabel('Epochs')
plt.legend()
plt.grid(True, linestyle='--')
plt.show()

# Visual 2: Forecast Precision (Zoom-in)
plt.figure(figsize=(15, 6))
plt.plot(inv_y[500:800], label='Actual
Pollution', color='#1f77b4')
plt.plot(inv_yhat[500:800], label='Tuned LSTM
Prediction', color='#d62728', linestyle='--')
plt.title('Prediction Precision (300 Hour
Zoom)')
plt.ylabel('PM2.5 Level')
plt.legend()
plt.show()

# Visual 3: Residual Analysis (Error
Distribution)
residuals = inv_y - inv_yhat
plt.figure(figsize=(10, 5))
plt.hist(residuals, bins=50, color='teal',
edgecolor='black', alpha=0.7)
plt.title('Distribution of Prediction Errors
(Residuals)')
plt.xlabel('Error Magnitude (Difference from
Reality)')
plt.ylabel('Frequency')
plt.show(
```

Output:

```
Epoch 22/50
548/548 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - loss: 0.0145 - val_loss: 0.0135 - learning_rate: 6.250
274/274 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step

==============================
   OPTIMIZED MODEL RESULTS
==============================
1. RMSE: 24.2706
2. R-Squared Accuracy: 0.9327 (or 93.27%)
```







This improved code builds a more advanced air pollution prediction system. It looks at the past 3 hours instead of 1, and uses a deeper, stacked LSTM model for better learning. It also automatically adjusts learning speed and stops early to avoid overtraining, then evaluates accuracy and visualizes prediction errors clearly.

Technical Justification for Improvement:

1. Temporal Depth (Lookback): By increasing the lookback from 1 hour to 3 hours, the model gained "contextual memory," allowing it to understand the momentum of weather changes rather than just reacting to immediate inputs.
2. Stacked LSTM Layers: Adding a second LSTM layer (100 units → 50 units) allowed the network to learn a hierarchical representation of the data—where the first layer extracts raw temporal features and the second layer extracts higher-level interactions.
3. Learning Rate Scheduling: The use of ReduceLROnPlateau allowed the model to "slow down" its learning as it approached the global minimum, preventing it from overshooting the optimal weights.
4. Dropout Regularization: Adding a 20% Dropout layer reduced the gap between training and validation loss, ensuring the R^2 increase was due to better generalization rather than overfitting.

Conclusion:

By mastering the LSTM's gating architecture and temporal dependencies, we've effectively transformed chaotic atmospheric fluctuations into high-precision predictive signals, achieving a **93.27%** synergy between raw meteorological data and future-state forecasting.