Temperature Anomaly Detection

This project,*Temperature Anomaly Detection*, has been carried out under the guidance of **Dr.J.V.Satyanarayana, Sc 'G' of DEAIS {RCI}**. The project has been reviewed and approved as part of our work in temperature anomaly detection.

By students from University College Of Engineering(OU)-CSE(AIML) Dept

K. Jithendra Ganesh

D. Likhitha

Guide's Name: **Dr.J.V.Satyanarayana**

Signature:

Abstract :

Weather anomalies such as sudden rainfall, extreme temperature fluctuations, or unusual humidity levels can lead to significant disruptions in agriculture, transportation, and infrastructure. Traditional methods of anomaly detection often fail to accurately capture non-linear and temporal dependencies in weather data. With the rise in deep learning models capable of handling sequential data, it's now possible to develop systems that learn patterns from historical data and detect deviations more effectively.

In this project, we propose a lightweight and memory-efficient approach to detect anomalies in weather data using LSTM-based Autoencoders and a Stacked Transformer with CNN (STOC) model. The problem statement provided by J.V. Satyanarayana Sir guided us to explore various anomaly detection techniques and assess their applicability on different data granularities—monthly, weekly, and yearly. Weekly data was chosen as it provides a good balance between trend visibility and dataset size.

The models are trained on historical weather data comprising parameters such as temperature, rainfall, humidity, and wind speed. Anomalies are identified by calculating the reconstruction error of sequences and setting a statistical threshold. This system can serve as an early warning tool for meteorological departments and other stakeholders.

Introduction:

Anomaly detection refers to identifying patterns in data that do not conform to expected behavior. In the context of weather, anomalies can include sudden changes in temperature, unseasonal rainfall, or unusual humidity levels. These anomalies, if detected early, can help prevent losses in agriculture, manage power consumption, and aid in disaster management.

Traditional time series forecasting techniques like ARIMA and simple statistical methods fall short in capturing complex patterns and non-linear relationships. Deep learning models, especially sequence-based models like LSTM and Transformers, have the capability to learn such patterns effectively from large datasets.

This project investigates the performance of two powerful deep learning models—LSTM Autoencoders and a custom STOC (Stacked Transformer with Convolution) model. These models are chosen for their efficiency in modeling sequential data and their adaptability to various data volumes. We specifically focus on building lightweight versions of these models to ensure that they can be executed on systems with limited hardware capabilities.

The project is conducted in stages—data collection and cleaning, preprocessing, model building, training, testing, and visualization. Each stage is critically analyzed and optimized for performance and efficiency. We also focus on reducing memory consumption to allow deployment on low-resource systems.

Libraries Used in Project :

The following Python libraries and frameworks were used throughout the project to implement various functionalities:

- **NumPy**: For numerical operations, array manipulation, and mathematical functions.
- **Pandas**: Essential for reading, filtering, and preprocessing time-series data.
- **Matplotlib and Seaborn**: Visualization libraries used to plot time-series graphs, reconstruction error plots, and anomaly points.
- **Scikit-learn**: Used for data normalization, model evaluation metrics, and preprocessing tools.
- **TensorFlow/Keras**: Core framework for building and training the LSTM Autoencoder and STOC models.
- **Google Colab / Jupyter Notebook**: Used as the development environment for writing and testing code.
- **Statsmodels**: Occasionally used for statistical analysis of time series.

Each of these libraries was chosen for their performance, ease of integration, and support for deep learning or time series data analysis. TensorFlow was selected over PyTorch for its memory optimization in mobile or edge deployment scenarios.

Dataset:

The dataset used in this project is a multivariate time-series dataset containing weekly weather records. Each record includes the following fields:

- **Temperature** (in Celsius)
- **Humidity** (in percentage)
- **Rainfall** (in millimeters)
- **Wind Speed** (in km/h)
- **Atmospheric Pressure** (in hPa)

Source:

- NOAA (National Oceanic and Atmospheric Administration)
- Kaggle-based weather datasets

Data Granularity:

- **Monthly**: Not detailed enough for early anomaly detection.
- **Yearly**: Too sparse and ineffective for machine learning.
- **Weekly**: Chosen for optimal resolution and consistency.

Data was recorded over several years to capture seasonal patterns and allow the models to learn from both normal and extreme variations. This wide range of data points enables the detection of both minor and major anomalies.

The dataset used in this project is a multivariate time-series dataset containing weekly weather records. Each record includes the following fields:

- **Temperature** (in Celsius)
- **Humidity** (in percentage)
- **Rainfall** (in millimeters)
- **Wind Speed** (in km/h)
- **Atmospheric Pressure** (in hPa)

Source:

- NOAA (National Oceanic and Atmospheric Administration)
- Kaggle-based weather datasets

Data Granularity:

- **Monthly**: Not detailed enough for early anomaly detection.
- **Yearly**: Too sparse and ineffective for machine learning.
- **Weekly**: Chosen for optimal resolution and consistency.

Data was recorded over several years to capture seasonal patterns and allow the models to learn from both normal and extreme variations. This wide range of data points enables the detection of both minor and major anomalies.

Preprocessing Process Done on Data

Effective preprocessing is critical in time-series anomaly detection. The following steps were taken:

1. **Handling Missing Values**: Forward-fill and interpolation techniques were used to handle missing entries in the dataset.
2. **Outlier Detection**: Visual inspection and Z-score method were used to identify and smooth obvious outliers.
3. **Normalization**: Min-Max scaling was applied to bring all features into a [0, 1] range for faster convergence during training.
4. **Sliding Window Technique**: Input sequences were created using a 50-timestep sliding window to generate meaningful time slices.
5. **Train-Test Split**: An 80-20 split was used for training and testing datasets, ensuring the test set included known anomalies.
6. **Sequence Reshaping**: Input sequences were reshaped into 3D tensors of the form (samples, timesteps, features) as required by LSTM and Transformer architectures.

This preprocessing pipeline ensured clean, normalized, and well-structured data suitable for deep learning models.

Methods Used and Their Architecture

Two deep learning methods were applied:

1. LSTM Autoencoder

- Learns a compressed representation of the input sequence.
- Attempts to reconstruct the input from this representation.
- Anomalies are identified when reconstruction error exceeds a threshold.

Architecture:

1. **Encoder:**
   - Input Layer: Shape = `(batch_size, 12, 6)` (12 time steps, 6 features)
   - LSTM Layer 1: 64 units, `return_sequences=True`
   - LSTM Layer 2: 32 units, `return_sequences=False`
2. **Decoder:**
   - Repeat Vector: Repeats the context vector for 12 time steps
   - LSTM Layer 1: 32 units, `return_sequences=True`
   - LSTM Layer 2: 64 units, `return_sequences=True`
3. **Output:**
   - TimeDistributed Dense Layer: 6 units (applied at each time step)
   - Activation: Linear (to match original input features)

2. Transformers

- A lightweight Transformer encoder followed by a 1D CNN block.
- Captures both long-term and local temporal patterns.
- Efficient in memory-constrained environments.

Architecture:

1. **Input Layer**
    - Shape: `(batch_size, sequence_length, feature_dim)` = `(None, 12, 6)`
2. **Positional Encoding (Learned)**
    - Shape: `(12, 64)`
    - Learnable embedding added to each timestep to retain order of sequence.
3. **Embedding Layer**
    - Dense projection from 6 → 64 dimensions
    - Shape: `(None, 12, 64)`
4. **Transformer Encoder Block (Single Block)**
   (This block is repeated only once to reduce memory use)
    - **Layer Normalization**
      Applied before attention (pre-norm)
      Shape: `(None, 12, 64)`

- **Multi-Head Self-Attention**
  - Heads: 1
  - Key/Query/Value dimension: 64
  - Shape: `(None, 12, 64)`
- **Add & Norm (Residual Connection)**
  Adds input to attention output and normalizes
  Shape: `(None, 12, 64)`
- **Feed-Forward Network**
  - Dense Layer 1: 64 units + ReLU
  - Dropout: 0.1
  - Dense Layer 2: 64 units
  - Shape: `(None, 12, 64)`
- **Add & Norm (Residual)**
  Adds attention block output to FFN output
  Shape: `(None, 12, 64)`

5. **Output Projection**
   - Dense Layer: 6 units (to match input feature dimension)
   - Output Shape: `(None, 12, 6)`

Both models use MSE loss and Adam optimizer for training.

Model Layer Configuration

LSTM Autoencoder:

- Input Shape: (50, 1)
- Encoder:
    - LSTM(128), return_sequences=True
    - LSTM(64)
- Latent Representation: Dense(32, activation='relu')
- Decoder:
    - RepeatVector(50)
    - LSTM(64, return_sequences=True)
    - LSTM(128, return_sequences=True)
- Output: TimeDistributed(Dense(1))

Transformer:

- Input: Weekly sequences reshaped for attention input
- Embedding Layer: 64 dimensions
- Transformer Encoder Block:
    - Multi-head Attention: 1 head
    - Feed-forward Network: 64 units
    - Dropout: 0.1
- Dense Output Layer: Linear projection to original input shape

Memory efficiency was prioritized by reducing the number of layers and avoiding unnecessary tensor reshaping.

Results of the Project

Evaluation:

- Models were evaluated using MSE on validation data.
- Anomaly threshold set using 95th percentile of training error.

Performance:

- **LSTM Autoencoder**: High accuracy, slow training
- **Transformer Model**: Faster, more efficient, similar accuracy

Visual Output:

- Plotted original vs reconstructed sequences
- Anomalies highlighted with red dots on line plots

Findings:

- Weekly weather data offered ideal balance
- Transformer model detected anomalies with minimal false positives
- Ideal for edge deployment

Use Cases:

- Weather forecasting
- Disaster early warning systems
- Agriculture and irrigation planning

Hardware and Software Requirements

Hardware Requirements:

Component	Specification
CPU	Intel i3 / Ryzen 3
RAM	4 GB minimum (8 GB ideal)
GPU	Not mandatory
Disk Storage	Minimum 2 GB free space

Software Requirements:

Component	Specification
OS	Windows/Linux/macOS
Programming	Python 3.8+
Libraries	TensorFlow, Keras, NumPy, Pandas
IDE	Google Colab / Jupyter Notebook