Comprehensive Analysis of Anomaly Detection Techniques for Dynamic and Graph-Based Data

Charlie Abela
Department of Artificial Intelligence

Contents

1 Introduction to Dynamic Data Anomaly Detection

Time series data constitutes a fundamental data structure across numerous domains, defined by sequences of observations indexed chronologically where the temporal ordering conveys critical contextual information. The ubiquity of such data is evident in financial markets (high-frequency trading data, stock prices, market indicators), industrial Internet of Things deployments (sensor readings from manufacturing equipment, smart grid measurements, supply chain metrics), information technology infrastructure (server performance metrics, network traffic patterns, application response times), healthcare monitoring systems (continuous ECG recordings, blood glucose measurements, patient vital signs), and meteorological observations (temperature readings, precipitation levels, atmospheric pressure). The temporal dimension in such datasets introduces complex dependencies that static analysis methodologies inevitably fail to capture, as temporal context fundamentally alters the interpretation of values-a server experiencing 95% CPU utilization at 3 PM during a scheduled batch processing operation represents normal behavior, while the same measurement at 3 AM might indicate a serious security breach or system failure.

The accurate detection of anomalies in time series data presents substantial technical challenges stemming from several inherent characteristics of temporal data. Temporal dependencies create complex patterns across various time scales, from short-term correlations between adjacent observations to long-term seasonal or cyclical patterns spanning days, weeks, or years. Non-stationarity manifests as evolving statistical properties including changing means, variances, and correlation structures. Many real-world time series exhibit complex seasonality patterns (daily, weekly, monthly) that must be distinguished from genuine anomalies. Additionally, temporal data often contains intrinsic noise that must be differentiated from meaningful signal deviations.

1.1 Types of Anomalies in Time Series

When analyzing temporal data, practitioners encounter three principal categories of anomalies, each requiring distinct detection approaches. Point anomalies (also termed global anomalies) represent individual observations that significantly deviate from the overall data distribution, independent of their temporal context. Consider an industrial temperature sensor that typically reports values between 50-70°C suddenly recording 150°C-such a reading constitutes a clear point anomaly that even simple statistical methods can readily identify. These anomalies may indicate sensor malfunctions, recording errors, or genuine but extreme events.

Contextual anomalies (sometimes called conditional anomalies) manifest as observations that appear anomalous only when considered within their specific temporal context. For instance, a retail website experiencing 10,000 users at 2 PM on a typical weekday represents normal traffic, whereas the same traffic volume at 2 AM would be highly suspicious and potentially indicate a distributed denial-of-service attack. Similarly, body temperature of 37.5°C might be normal during physical exercise but anomalous during rest. Detecting contextual anomalies requires models that effectively incorporate temporal context, seasonality, and domain-specific normal patterns.

Collective anomalies comprise sequences of observations that, when considered together, form an unusual pattern despite individual points potentially appearing normal when examined in isolation. A classic example comes from cardiology, where a sequence of ECG readings might individually fall within normal ranges but collectively form an arrhythmic pattern indicative of cardiac dysfunction. Similarly, in cybersecurity, a sequence of individually legitimate commands executed in an unusual order might represent an attack pattern. Such anomalies necessitate methods capable of modeling sequential patterns and dependencies rather than examining points independently.

1.2 Statistical Methods for Time Series Anomaly Detection

Statistical approaches to anomaly detection in time series data leverage established mathematical frameworks to identify observations that deviate significantly from expected patterns. These methods often serve as both standalone detection techniques and as fundamental components within more complex systems.

1.2.1 Window-Based Statistical Approaches

Window-based techniques operate by defining a sliding window of observations within the time series and calculating statistical metrics within this window to establish a local norm against which current observations are evaluated. This approach naturally adapts to gradual changes in the underlying data distribution while maintaining sensitivity to sudden anomalous deviations.

Formally, for a time series $\{x_1, x_2, \ldots, x_T\}$ and a window of size w, we compute the window mean μ_t and standard deviation σ_t at time t as:

$$\mu_t = \frac{1}{w} \sum_{i=t-w}^{t-1} x_i \tag{1}$$

$$\sigma_t = \sqrt{\frac{1}{w} \sum_{i=t-w}^{t-1} (x_i - \mu_t)^2} \tag{2}$$

An observation x_t is classified as anomalous if it deviates from this local distribution by more than k standard deviations, where k is a sensitivity parameter typically set between 2 and 3:

$$|x_t - \mu_t| > k \cdot \sigma_t \tag{3}$$

To illustrate this approach with a concrete example, consider a server CPU utilization time series with values [10%, 12%, 11%, 13%, 10%, 30%, 12%]. Using a window size $w = 5$ and sensitivity threshold $k = 3$, we evaluate the sixth value (30%) against the preceding window [10%, 12%, 11%, 13%, 10%]. The window mean is calculated as $\mu_6 = \frac{10+12+11+13+10}{5} = 11.2\%$, and the standard deviation as $\sigma_6 = \sqrt{\frac{(10-11.2)^2+(12-11.2)^2+(11-11.2)^2+(13-11.2)^2+(10-11.2)^2}{5}} = 1.16\%$. The detection

threshold becomes $11.2\% \pm 3 \times 1.16\% = [7.72\%, 14.68\%]$. Since 30% falls well outside this range (30% ¿ 14.68%), we correctly identify this observation as anomalous.

Various refinements to this basic approach include:

- **Median Absolute Deviation (MAD)**: Replaces mean/standard deviation with median/MAD to improve robustness against outliers and non-Gaussian distributions:

$$\text{median}_t = \text{median}(\{x_{t-w}, x_{t-w+1}, \ldots, x_{t-1}\}) \tag{4}$$

$$\text{MAD}_t = \text{median}(\{|x_{t-w} - \text{median}_t|, \ldots, |x_{t-1} - \text{median}_t|\}) \tag{5}$$

$$\text{Anomaly if: } |x_t - \text{median}_t| > k \cdot 1.4826 \cdot \text{MAD}_t \tag{6}$$

  where 1.4826 is a scaling factor for consistency with standard normal distribution.
- **Exponentially Weighted Moving Statistics**: Assigns exponentially decreasing weights to older observations, giving more importance to recent data:

$$\mu_t = \alpha \cdot x_{t-1} + (1 - \alpha) \cdot \mu_{t-1} \tag{7}$$

$$\sigma_t^2 = \alpha \cdot (x_{t-1} - \mu_t)^2 + (1 - \alpha) \cdot \sigma_{t-1}^2 \tag{8}$$

  where $\alpha \in (0, 1)$ is the smoothing factor.
- **Adaptive Thresholding**: Dynamically adjusts the sensitivity parameter k based on the characteristics of the time series or specific application requirements.

The following code demonstrates an implementation of these window-based approaches:

Listing 1 Window-Based Statistical Anomaly Detection

```
import numpy as np
import pandas as pd
from scipy import stats

def moving_average_detector(time_series, window_size=20, k=3, method='std'):
    """
    Detect anomalies using moving window statistics.

    Parameters:
    -----------
    time_series : array-like
        The input time series data
    window_size : int
        The size of the sliding window
    k : float
        The threshold multiplier (sensitivity)
    method : str
        'std' for standard deviation, 'mad' for median absolute deviation

    Returns:
    --------
    anomalies : array-like
        Boolean array where True indicates anomalies
    """
    ts = np.asarray(time_series)
    anomalies = np.zeros(len(ts), dtype=bool)

    # Need at least window_size observations before detection starts
    if len(ts) <= window_size:
```

```
        return anomalies

    # Initialize with NaN for the first window_size elements
    anomalies[:window_size] = False

    for t in range(window_size, len(ts)):
        # Get the window
        window = ts[t-window_size:t]

        if method == 'std':
            # Calculate mean and standard deviation
            window_mean = np.mean(window)
            window_std = np.std(window)

            # Check if current point is an anomaly
            if abs(ts[t] - window_mean) > k * window_std:
                anomalies[t] = True

        elif method == 'mad':
            # Calculate median and MAD
            window_median = np.median(window)
            window_mad = stats.median_abs_deviation(window, scale=1)  # Unscaled
                 MAD

            # Apply consistency constant for normal distribution
            window_mad_scaled = 1.4826 * window_mad

            # Check if current point is an anomaly
            if abs(ts[t] - window_median) > k * window_mad_scaled:
                anomalies[t] = True

    return anomalies

# Example usage
cpu_utilization = [10, 12, 11, 13, 10, 30, 12, 11, 10, 13]
anomalies_std = moving_average_detector(cpu_utilization, window_size=5, k=3, method
    ='std')
anomalies_mad = moving_average_detector(cpu_utilization, window_size=5, k=3, method
    ='mad')

print("Standard deviation method anomalies:", np.where(anomalies_std)[0].tolist())
print("MAD method anomalies:", np.where(anomalies_mad)[0].tolist())
```

While conceptually straightforward and computationally efficient, window-based approaches have several limitations. They struggle with multiple seasonality patterns, require careful window size selection (too small increases false positives, too large reduces sensitivity to local anomalies), and perform poorly on highly non-stationary data where statistical properties evolve rapidly.

1.2.2 ARIMA and Forecast-Based Methods

Forecast-based anomaly detection represents a sophisticated extension of statistical approaches, operating on the principle that anomalies are observations that significantly deviate from their predicted values. These methods employ time series forecasting models to predict the expected value at each time point, then compare the actual observation against this prediction, flagging significant deviations as potential anomalies.

AutoRegressive Integrated Moving Average (ARIMA) models constitute a classical framework for time series forecasting, combining three key components: autoregression (AR), which models the dependency between an observation and a specified number

of lagged observations; differencing (I), which transforms a non-stationary series to stationarity by computing differences between consecutive observations; and moving average (MA), which models the dependency between an observation and a residual error from a moving average model applied to lagged observations.

The ARIMA(p,d,q) model is formally defined as:

$$(1 - \sum_{i=1}^{p} \phi_i L^i)(1 - L)^d X_t = (1 + \sum_{i=1}^{q} \theta_i L^i)\varepsilon_t \tag{9}$$

where:

- p is the order of the autoregressive model
- d is the degree of differencing
- q is the order of the moving average model
- L is the lag operator
- ϕ_i are the parameters of the autoregressive part
- θ_i are the parameters of the moving average part
- ε_t is white noise

For anomaly detection, an ARIMA model is first trained on historical data presumed to be normal. For each new observation x_t, the model generates a forecast $\hat{x}_t$ along with a prediction interval. The observation is flagged as anomalous if it falls outside this prediction interval, typically set at 95% or 99% confidence:

$$x_t \notin [\hat{x}_t - z_{\alpha/2} \cdot \sigma_t, \hat{x}_t + z_{\alpha/2} \cdot \sigma_t] \tag{10}$$

where $z_{\alpha/2}$ is the critical value from the standard normal distribution corresponding to the desired confidence level (e.g., 1.96 for 95% confidence), and σ_t is the standard error of the forecast.

Alternatively, we can define an anomaly score based on the standardized residual:

$$\text{score}(x_t) = \frac{|x_t - \hat{x}_t|}{\sigma_t} \tag{11}$$

with values exceeding some threshold (typically 2 or 3) classified as anomalies.

Seasonal ARIMA (SARIMA) extends the base model to incorporate seasonal patterns:

$$(1-\sum_{i=1}^{p} \phi_i L^i)(1-\sum_{i=1}^{P} \Phi_i L^{is})(1-L)^d(1-L^s)^D X_t = (1+\sum_{i=1}^{q} \theta_i L^i)(1+\sum_{i=1}^{Q} \Theta_i L^{is})\varepsilon_t \tag{12}$$

where s is the seasonal period, and P, D, and Q are the seasonal equivalents of p, d, and q.

The following code demonstrates ARIMA-based anomaly detection:

Listing 2 ARIMA-Based Anomaly Detection

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```python
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import adfuller

def test_stationarity(timeseries):
    """Test for stationarity using Augmented Dickey-Fuller test"""
    result = adfuller(timeseries)
    print(f'ADF Statistic: {result[0]}')
    print(f'p-value: {result[1]}')
    print(f'Critical Values:')
    for key, value in result[4].items():
        print(f'\t{key}: {value}')

    # If p-value is less than 0.05, the series is stationary
    return result[1] < 0.05

def arima_anomaly_detector(time_series, train_size=0.7, order=(1,1,1), alpha=0.05):
    """
    Detect anomalies using ARIMA forecasting.

    Parameters:
    -----------
    time_series : array-like
        The input time series data
    train_size : float
        Proportion of data to use for training
    order : tuple
        ARIMA model order (p,d,q)
    alpha : float
        Significance level for prediction intervals (e.g., 0.05 for 95% confidence)

    Returns:
    --------
    anomalies : array-like
        Boolean array where True indicates anomalies
    forecast : array-like
        Forecasted values for the test period
    conf_int : array-like
        Confidence intervals for the forecast
    """
    ts = np.asarray(time_series)
    n = len(ts)
    train_idx = int(train_size * n)

    # Split into train and test
    train_data = ts[:train_idx]
    test_data = ts[train_idx:]

    # Check stationarity of differenced series if d > 0
    if order[1] > 0:
        diff_data = np.diff(train_data, n=order[1])
        is_stationary = test_stationarity(diff_data)
        print(f"Differenced series stationary: {is_stationary}")

    # Fit ARIMA model
    model = ARIMA(train_data, order=order)
    model_fit = model.fit()
    print(model_fit.summary())

    # Forecast for test period with confidence intervals
    forecast_result = model_fit.get_forecast(steps=len(test_data))
    forecast = forecast_result.predicted_mean
    conf_int = forecast_result.conf_int(alpha=alpha)

    # Detect anomalies (points outside confidence interval)
    lower_bound = conf_int[:, 0]
    upper_bound = conf_int[:, 1]
    anomalies = (test_data < lower_bound) | (test_data > upper_bound)

```

```
    # Visualize results
    plt.figure(figsize=(12, 6))
    plt.plot(range(n), ts, 'b-', label='Original Data')
    plt.plot(range(train_idx, n), forecast, 'r-', label='Forecast')
    plt.fill_between(range(train_idx, n), lower_bound, upper_bound, color='pink',
        alpha=0.3, label='95% Confidence Interval')
    plt.scatter(np.array(range(train_idx, n))[anomalies], test_data[anomalies],
        color='red', s=100, label='Anomalies', zorder=5)
    plt.legend()
    plt.title('ARIMA-based Anomaly Detection')
    plt.show()

    return anomalies, forecast, conf_int

# Example usage
np.random.seed(42)
# Generate synthetic time series with trend, seasonality, and anomalies
t = np.arange(0, 200)
trend = 0.05 * t
seasonality = 10 * np.sin(2 * np.pi * t / 50)
noise = np.random.normal(0, 1, 200)
ts = trend + seasonality + noise

# Insert anomalies
ts[80] += 15  # Point anomaly
ts[150:155] += 10  # Collective anomaly

# Detect anomalies
anomalies, forecast, conf_int = arima_anomaly_detector(ts, order=(2,1,2))
print(f"Detected anomalies at indices: {np.where(anomalies)[0] + int(0.7*len(ts))}"
    )
```

Forecast-based methods like ARIMA offer several advantages: they naturally handle trends and seasonality through model specification, provide probabilistic prediction intervals allowing for principled threshold setting, and can capture complex temporal dependencies. However, they also present challenges: ARIMA assumes linear relationships between variables, requires careful parameter selection (p,d,q values), becomes computationally intensive for long series, and may struggle with complex non-linear patterns or abrupt regime changes.

Additional forecast-based methods include:

- **Exponential Smoothing State Space Models (ETS)**: Decompose series into trend, seasonal, and error components with exponentially weighted observations.
- **Prophet**: Developed by Facebook, handles multiple seasonality patterns and automatically detects changepoints in time series.
- **GARCH Models**: Particularly useful for series with time-varying volatility, common in financial data.

1.2.3 Seasonal Decomposition

Seasonal decomposition approaches separate a time series into distinct components-typically trend, seasonality, and residuals-and identify anomalies by analyzing the residual component. This methodology proves particularly effective for time series with strong seasonal patterns where direct analysis of the raw data might miss anomalies masked by expected seasonal variations.

The classical decomposition model expresses a time series y_t as:

$$y_t = T_t + S_t + R_t \tag{13}$$

for additive decomposition, or:

$$y_t = T_t \times S_t \times R_t \tag{14}$$

for multiplicative decomposition, where T_t represents the trend component, S_t the seasonal component, and R_t the residual or irregular component.

The STL (Seasonal and Trend decomposition using Loess) method offers a robust decomposition approach. After decomposition, anomaly detection focuses on the residual component, flagging observations where residuals significantly deviate from their expected distribution. A common approach defines anomalies as residuals exceeding k standard deviations from zero:

$$|R_t| > k \cdot \sigma_R \tag{15}$$

where σ_R is the standard deviation of the residuals and k is typically set between 2 and 3.

Residual-based anomaly scores can also be calculated as:

$$\text{score}(y_t) = \frac{|R_t|}{\sigma_R} \tag{16}$$

providing a standardized measure of deviation.

Seasonal decomposition methods excel in scenarios with clear seasonal patterns and gradual trends. They effectively separate expected seasonal variations from genuine anomalies and provide interpretable decompositions that aid in understanding the underlying patterns in the data. However, they may struggle with complex, non-linear trends, multiple overlapping seasonal patterns, or abrupt changes in the underlying data distribution.

Advanced variations include MSTL (Multiple STL) for handling multiple seasonal patterns and robust decomposition methods that are less sensitive to extreme values and outliers.

1.3 Neural Networks for Time Series

Traditional statistical methods, while powerful and interpretable, often struggle with highly complex, non-linear patterns frequently present in real-world time series data. Neural network-based approaches offer significant advantages in these scenarios due to their capacity to learn intricate temporal dependencies directly from data without requiring explicit model specification.

1.3.1 Feedforward Neural Networks

Feedforward Neural Networks (FFNNs) represent the fundamental architecture in deep learning, consisting of an input layer that receives features, multiple hidden layers that transform the data through learned weights, and an output layer that produces the final prediction or classification. Despite not being specifically designed for

sequential data, FFNNs can be adapted for time series analysis through clever feature engineering.

For time series anomaly detection, FFNNs typically employ a sliding window approach, where a window of w previous observations $[x_{t-w}, x_{t-w+1}, \ldots, x_{t-1}]$ is used to predict the current value x_t. This window is flattened into a feature vector of length w that serves as input to the network. The network then maps this input to a predicted value $\hat{x}_t$ through a series of non-linear transformations:

$$h^{(1)} = \sigma(W^{(1)} \cdot [x_{t-w}, x_{t-w+1}, \ldots, x_{t-1}] + b^{(1)}) \tag{17}$$

$$h^{(2)} = \sigma(W^{(2)} \cdot h^{(1)} + b^{(2)}) \tag{18}$$

$$\vdots \tag{19}$$

$$h^{(L)} = \sigma(W^{(L)} \cdot h^{(L-1)} + b^{(L)}) \tag{20}$$

$$\hat{x}_t = W^{(L+1)} \cdot h^{(L)} + b^{(L+1)} \tag{21}$$

where $W^{(l)}$ and $b^{(l)}$ are the weights and biases at layer l, and σ is a non-linear activation function such as ReLU (Rectified Linear Unit): $\sigma(z) = \max(0, z)$.

Anomaly detection then proceeds by comparing the predicted value $\hat{x}_t$ with the actual observation x_t, flagging significant deviations as anomalies:

$$|x_t - \hat{x}_t| > \tau \tag{22}$$

where τ is a threshold that can be set based on the distribution of prediction errors on a validation set.

The following code implements a FFNN for time series anomaly detection:

Listing 3 Feedforward Neural Network for Time Series Anomaly Detection

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

def create_sequences(data, window_size):
    """
    Create sequences from time series data for supervised learning.

    Parameters:
    -----------
    data : array-like
        Input time series data
    window_size : int
        Size of the sliding window

    Returns:
    --------
    X : array
        Input sequences (features)
    y : array
        Target values (labels)
```

```
    """
    X, y = [], []
    for i in range(len(data) - window_size):
        X.append(data[i:i+window_size])
        y.append(data[i+window_size])
    return np.array(X), np.array(y)

def ffnn_anomaly_detector(time_series, window_size=10, train_size=0.7,
                          validation_size=0.15, threshold_percentile=99):
    """
    Detect anomalies using a Feedforward Neural Network.

    Parameters:
    -----------
    time_series : array-like
        Input time series data
    window_size : int
        Size of the sliding window
    train_size : float
        Proportion of data to use for training
    validation_size : float
        Proportion of data to use for validation
    threshold_percentile : float
        Percentile used to set anomaly threshold

    Returns:
    --------
    anomalies : array-like
        Boolean array where True indicates anomalies
    predictions : array-like
        Predicted values for the entire series
    threshold : float
        The anomaly detection threshold
    """
    # Normalize data
    scaler = StandardScaler()
    ts_normalized = scaler.fit_transform(np.array(time_series).reshape(-1, 1)).
        flatten()

    # Create sequences
    X, y = create_sequences(ts_normalized, window_size)

    # Split data
    n_samples = len(X)
    train_end = int(train_size * n_samples)
    val_end = int((train_size + validation_size) * n_samples)

    X_train, y_train = X[:train_end], y[:train_end]
    X_val, y_val = X[train_end:val_end], y[train_end:val_end]
    X_test, y_test = X[val_end:], y[val_end:]

    # Build model
    model = Sequential([
        Dense(64, activation='relu', input_shape=(window_size,)),
        Dropout(0.2),  # Regularization to prevent overfitting
        Dense(32, activation='relu'),
        Dropout(0.2),
        Dense(1)  # Output layer for regression
    ])

    # Compile model
    model.compile(optimizer='adam', loss='mse')

    # Early stopping to prevent overfitting
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=10,
        mode='min',
```

```
        restore_best_weights=True
    )

    # Train model
    history = model.fit(
        X_train, y_train,
        epochs=100,
        batch_size=32,
        validation_data=(X_val, y_val),
        callbacks=[early_stopping],
        verbose=0
    )

    # Plot training history
    plt.figure(figsize=(10, 6))
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Model Training History')
    plt.ylabel('Loss (MSE)')
    plt.xlabel('Epoch')
    plt.legend()
    plt.show()

    # Predict on all data and calculate errors
    X_all = X
    y_all = y
    y_pred = model.predict(X_all).flatten()
    errors = np.abs(y_all - y_pred)

    # Determine threshold from validation set errors
    val_errors = np.abs(y_val - model.predict(X_val).flatten())
    threshold = np.percentile(val_errors, threshold_percentile)

    # Detect anomalies
    anomalies = errors > threshold

    # Map back to original time series length
    full_anomalies = np.zeros(len(time_series), dtype=bool)
    full_predictions = np.zeros(len(time_series)) * np.nan

    # First window_size elements cannot be predicted with this approach
    full_anomalies[window_size:] = anomalies

    # Denormalize predictions
    predictions_denorm = scaler.inverse_transform(
        y_pred.reshape(-1, 1)).flatten()
    full_predictions[window_size:] = predictions_denorm

    # Visualize results
    plt.figure(figsize=(12, 6))
    plt.plot(time_series, label='Original')
    plt.plot(full_predictions, 'r--', alpha=0.7, label='Predicted')
    plt.scatter(
        np.where(full_anomalies)[0],
        np.array(time_series)[full_anomalies],
        color='red', s=80, marker='o', label='Anomalies'
    )

    plt.title(f'FFNN Anomaly Detection (Window Size: {window_size})')
    plt.legend()
    plt.show()

    return full_anomalies, full_predictions, threshold

# Example usage
np.random.seed(42)
# Generate synthetic time series with trend, seasonality, and anomalies
t = np.arange(0, 500)
```

```
trend = 0.05 * t
seasonality = 10 * np.sin(2 * np.pi * t / 50)
noise = np.random.normal(0, 1, 500)
ts = trend + seasonality + noise

# Insert anomalies
ts[150] += 15  # Point anomaly
ts[300:305] += 10  # Collective anomaly
ts[400:410] -= 15  # Extended anomaly

# Detect anomalies
anomalies, predictions, threshold = ffnn_anomaly_detector(
    ts, window_size=20, threshold_percentile=99)
print(f"Detected {sum(anomalies)} anomalies using threshold {threshold:.4f}")
```

Despite their utility, FFNNs exhibit significant limitations when applied to time series. They require fixed-size inputs, making them inflexible for variable-length sequences. The sliding window approach loses sequential information beyond the window size, limiting the network's ability to capture long-range dependencies. Additionally, FFNNs treat each time step independently within the window, ignoring the natural ordering of sequential data. These limitations motivated the development of specialized neural network architectures specifically designed for sequential data processing.

1.3.2 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks introduce a fundamental architectural innovation designed specifically for sequential data: self-looping connections that maintain an internal state (memory) across time steps. This internal state allows RNNs to naturally process sequences of variable length while preserving information about previous observations, making them substantially more suitable for time series analysis than feedforward networks.

The core operation of an RNN involves updating its hidden state based on both the current input and the previous hidden state:

$$h_t = \phi(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t + b_h) \tag{23}$$

where:

- $h_t \in \mathbb{R}^d$ is the hidden state at time step t
- $x_t \in \mathbb{R}^n$ is the input at time step t
- $W_{hh} \in \mathbb{R}^{d \times d}$ is the recurrent weight matrix for hidden-to-hidden connections
- $W_{xh} \in \mathbb{R}^{d \times n}$ is the weight matrix for input-to-hidden connections
- $b_h \in \mathbb{R}^d$ is the bias vector
- ϕ is an activation function, typically tanh or ReLU

The output at each time step is then computed as:

$$y_t = W_{hy} \cdot h_t + b_y \tag{24}$$

where $W_{hy} \in \mathbb{R}^{m \times d}$ is the output weight matrix, and $b_y \in \mathbb{R}^m$ is the output bias.

A critical insight in RNN design is parameter sharing across time steps-the same weights (W_{hh}, W_{xh}, W_{hy}) are used at each time step. This dramatically reduces

the number of parameters compared to feedforward networks processing sequences, enables handling of variable-length sequences, and allows the model to generalize across different positions in time.

For a concrete example, consider an RNN with a one-dimensional input, a two-dimensional hidden state, and randomly initialized weights:

$$W_{hh} = \begin{bmatrix} 0.3 & 0.4 \\ 0.2 & 0.5 \end{bmatrix} \tag{25}$$

$$W_{xh} = \begin{bmatrix} 0.8 \\ 0.7 \end{bmatrix} \tag{26}$$

$$b_h = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} \tag{27}$$

Given an initial hidden state $h_0 = [0, 0]^T$ and an input sequence $x = [1, 2, 3]$, the hidden state evolution proceeds as follows:

$$h_1 = \tanh\left(W_{hh} \cdot h_0 + W_{xh} \cdot x_1 + b_h\right) \tag{28}$$

$$= \tanh\left(\begin{bmatrix} 0.3 & 0.4 \\ 0.2 & 0.5 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.8 \\ 0.7 \end{bmatrix} \cdot 1 + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}\right) \tag{29}$$

$$= \tanh\left(\begin{bmatrix} 0.9 \\ 0.9 \end{bmatrix}\right) \tag{30}$$

$$= \begin{bmatrix} 0.716 \\ 0.716 \end{bmatrix} \tag{31}$$

For the second time step:

$$h_2 = \tanh\left(W_{hh} \cdot h_1 + W_{xh} \cdot x_2 + b_h\right) \tag{32}$$

$$= \tanh\left(\begin{bmatrix} 0.3 & 0.4 \\ 0.2 & 0.5 \end{bmatrix} \begin{bmatrix} 0.716 \\ 0.716 \end{bmatrix} + \begin{bmatrix} 0.8 \\ 0.7 \end{bmatrix} \cdot 2 + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}\right) \tag{33}$$

$$= \tanh\left(\begin{bmatrix} 0.3 \cdot 0.716 + 0.4 \cdot 0.716 + 0.8 \cdot 2 + 0.1 \\ 0.2 \cdot 0.716 + 0.5 \cdot 0.716 + 0.7 \cdot 2 + 0.2 \end{bmatrix}\right) \tag{34}$$

$$= \tanh\left(\begin{bmatrix} 2.30 \\ 2.30 \end{bmatrix}\right) \tag{35}$$

$$= \begin{bmatrix} 0.98 \\ 0.98 \end{bmatrix} \tag{36}$$

This process continues for subsequent time steps, with the hidden state retaining and integrating information from all previous inputs. For anomaly detection, RNNs can be trained to predict the next value in a sequence, with anomalies identified as

observations that significantly deviate from their predicted values-an approach conceptually similar to forecast-based methods but leveraging the sequential modeling power of RNNs.

The Vanishing Gradient Problem

Despite their conceptual elegance, vanilla RNNs suffer from a critical limitation known as the vanishing gradient problem, which severely restricts their ability to learn long-term dependencies in sequences. This problem emerges from the mathematical properties of backpropagation through time (BPTT), the algorithm used to train RNNs.

During BPTT, the gradient of the loss function with respect to the network weights at a particular time step depends on gradients from all future time steps. For a loss L at time step T, the gradient with respect to the recurrent weights W_{hh} involves a product chain over time steps:

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial y_T} \frac{\partial y_T}{\partial h_T} \frac{\partial h_T}{\partial h_t} \frac{\partial h_t}{\partial W_{hh}} \tag{37}$$

The term $\frac{\partial h_T}{\partial h_t}$ represents how the hidden state at time T is influenced by the hidden state at an earlier time t. This term expands to:

$$\frac{\partial h_T}{\partial h_t} = \prod_{i=t+1}^{T} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=t+1}^{T} W_{hh}^{\top} \text{diag}(\phi'(h_{i-1})) \tag{38}$$

where ϕ' is the derivative of the activation function. For common activation functions like tanh or sigmoid, the derivative is bounded by a small value (e.g., $\phi'(x) \leq 0.25$ for tanh at all points). Consequently, when multiplied repeatedly over many time steps, the gradient exponentially approaches zero:

$$\left\| \frac{\partial h_T}{\partial h_t} \right\| \leq \|W_{hh}\|^{T-t} \cdot 0.25^{T-t} \tag{39}$$

For long sequences where $T - t$ is large, the gradient effectively vanishes, preventing the network from learning dependencies between temporally distant observations. This limitation is particularly problematic for anomaly detection in time series, where patterns may depend on observations from many time steps earlier.

The following code demonstrates this vanishing gradient effect:

Listing 4 Demonstrating the Vanishing Gradient Problem in RNNs

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.callbacks import Callback

# Custom callback to track gradients
class GradientTracker(Callback):
    def __init__(self):
        self.gradients = []

```

```
    def on_batch_end(self, batch, logs=None):
        grads = self.model.optimizer.get_gradients(
            self.model.total_loss, self.model.trainable_weights)
        # Get the gradient
        norm = tf.norm(grads[0])
        self.gradients.append(norm.numpy())

def generate_sequence_data(seq_length=100, pattern_gap=10):
    """
    Generate synthetic sequence data that requires long-term memory.
    The target is determined by the input from pattern_gap steps back.
    """
    # Generate data
    X = np.zeros((1000, seq_length, 1))
    y = np.zeros((1000, 1))

    for i in range(1000):
        # Random binary sequence
        X[i, :, 0] = np.random.randint(0, 2, seq_length)
        # The target depends on an element pattern_gap steps back
        target_position = np.random.randint(0, seq_length - pattern_gap - 1)
        y[i, 0] = X[i, target_position, 0]

    return X, y

# Demonstrate vanishing gradient with two different sequence lengths
def demonstrate_vanishing_gradient():
    results = {}

    for seq_length in [20, 100]:
        print(f"\nTraining with sequence length {seq_length}")
        X, y = generate_sequence_data(seq_length=seq_length)

        # Build simple RNN model
        model = Sequential([
            SimpleRNN(32, return_sequences=False, input_shape=(seq_length, 1)),
            Dense(1, activation='sigmoid')
        ])

        model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),
                      loss='binary_crossentropy',
                      metrics=['accuracy'])

        # Track gradients
        gradient_tracker = GradientTracker()

        # Train for a few epochs
        history = model.fit(X, y, epochs=5, batch_size=32,
                            callbacks=[gradient_tracker], verbose=1)

        results[seq_length] = {
            'history': history.history,
            'gradients': gradient_tracker.gradients
        }

    # Plot gradient magnitudes
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.plot(results[20]['gradients'], label='Sequence Length: 20')
    plt.plot(results[100]['gradients'], label='Sequence Length: 100')
    plt.title('Gradient Magnitude During Training')
    plt.xlabel('Batch Updates')
    plt.ylabel('Gradient L2 Norm')
    plt.yscale('log')  # Log scale to better visualize differences
    plt.legend()
    plt.grid(True)

```

Fig. 1 LSTM cell architecture showing the cell state and three gates.

```
    plt.subplot(1, 2, 2)
    plt.plot(results[20]['history']['accuracy'], label='Seq Length 20')
    plt.plot(results[100]['history']['accuracy'], label='Seq Length 100')
    plt.title('Model Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.show()

    return results

gradient_results = demonstrate_vanishing_gradient()
```

This code example demonstrates how the gradient magnitude diminishes dramatically with increasing sequence length, making it difficult for the RNN to learn from longer sequences. The visualization clearly shows lower gradient magnitudes and worse performance for the longer sequence (100 time steps) compared to the shorter one (20 time steps), illustrating the practical impact of the vanishing gradient problem.

1.3.3 Long Short-Term Memory (LSTM) Networks

Long Short-Term Memory (LSTM) networks were specifically designed to address the vanishing gradient problem that plagues standard RNNs. Introduced by Hochreiter and Schmidhuber in 1997, LSTMs incorporate an ingenious architectural innovation: a "cell state" that acts as a highway for information flow through the network, protected by specialized gate mechanisms that regulate what information enters, remains, or exits this state.

The core mechanism of an LSTM cell consists of three gates-forget, input, and output-controlling information flow through the cell state:

Cell State and Gate Mechanisms

The cell state C_t serves as the primary memory component of the LSTM, designed to allow unimpeded gradient flow during backpropagation. Unlike hidden states in standard RNNs, the cell state is protected by carefully controlled multiplicative interactions, allowing it to maintain information over many time steps.

The forget gate determines what information should be discarded from the cell state, functioning as a "filter" for outdated or irrelevant information:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{40}$$

where σ is the sigmoid function, producing outputs between 0 (completely forget) and 1 (completely retain), W_f is the weight matrix for the forget gate, $[h_{t-1}, x_t]$ represents the concatenation of the previous hidden state and current input, and b_f is the bias term. When the sigmoid outputs values close to zero, the corresponding information in the cell state will be effectively erased, while values close to one preserve information.

The input gate controls what new information should be added to the cell state, serving as a second "filter" for incoming information:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{41}$$

Simultaneously, a candidate update vector $\tilde{C}_t$ is computed using a tanh activation, representing potential new values for the cell state:

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{42}$$

These components combine to update the cell state according to:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{43}$$

where $\odot$ represents element-wise multiplication. This equation demonstrates how the forget gate (f_t) controls what information is retained from the previous cell state (C_{t-1}), while the input gate (i_t) controls what new information ($\tilde{C}_t$) is added.

Finally, the output gate determines what information from the updated cell state should be exposed as the hidden state (output) of the LSTM cell:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{44}$$

The hidden state is then computed as:

$$h_t = o_t \odot \tanh(C_t) \tag{45}$$

This architecture allows LSTMs to selectively remember relevant information over many time steps while discarding irrelevant information, making them particularly effective for capturing long-range dependencies in sequences-a critical capability for anomaly detection in time series with complex patterns.

LSTM Step-by-Step Example

To illustrate the LSTM mechanism, consider a simple numerical example with a one-dimensional input and cell state:

$$x_t = [0.5] \text{ (current input)} \tag{46}$$
$$h_{t-1} = [0.2] \text{ (previous hidden state)} \tag{47}$$
$$C_{t-1} = [0.3] \text{ (previous cell state)} \tag{48}$$

With the following parameter values:

$$W_f = [0.7, 0.2], \quad b_f = [-0.3] \tag{49}$$
$$W_i = [0.4, 0.2], \quad b_i = [0.0] \tag{50}$$
$$W_C = [0.3, 0.4], \quad b_C = [0.0] \tag{51}$$
$$W_o = [0.5, 0.3], \quad b_o = [0.2] \tag{52}$$

The forget gate calculation is:

$$
\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) && (53)\\
&= \sigma([0.7, 0.2] \cdot [0.2, 0.5] + [-0.3]) && (54)\\
&= \sigma((0.7 \times 0.2) + (0.2 \times 0.5) - 0.3) && (55)\\
&= \sigma(0.14 + 0.1 - 0.3) && (56)\\
&= \sigma(-0.06) && (57)\\
&= 0.485 && (58)
\end{aligned}
$$

The input gate calculation is:

$$
\begin{aligned}
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) && (59)\\
&= \sigma([0.4, 0.2] \cdot [0.2, 0.5] + [0.0]) && (60)\\
&= \sigma((0.4 \times 0.2) + (0.2 \times 0.5)) && (61)\\
&= \sigma(0.08 + 0.1) && (62)\\
&= \sigma(0.18) && (63)\\
&= 0.545 && (64)
\end{aligned}
$$

The candidate cell state is:

$$
\begin{aligned}
\tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) && (65)\\
&= \tanh([0.3, 0.4] \cdot [0.2, 0.5] + [0.0]) && (66)\\
&= \tanh((0.3 \times 0.2) + (0.4 \times 0.5)) && (67)\\
&= \tanh(0.06 + 0.2) && (68)\\
&= \tanh(0.26) && (69)\\
&= 0.255 && (70)
\end{aligned}
$$

Now we update the cell state:

$$
\begin{aligned}
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t && (71)\\
&= 0.485 \times 0.3 + 0.545 \times 0.255 && (72)\\
&= 0.1455 + 0.139 && (73)\\
&= 0.2845 && (74)
\end{aligned}
$$

The output gate is:

$$
o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (75)
$$

$$= \sigma([0.5, 0.3] \cdot [0.2, 0.5] + [0.2]) \tag{76}$$
$$= \sigma((0.5 \times 0.2) + (0.3 \times 0.5) + 0.2) \tag{77}$$
$$= \sigma(0.1 + 0.15 + 0.2) \tag{78}$$
$$= \sigma(0.45) \tag{79}$$
$$= 0.611 \tag{80}$$

Finally, the hidden state is:

$$h_t = o_t \odot \tanh(C_t) \tag{81}$$
$$= 0.611 \times \tanh(0.2845) \tag{82}$$
$$= 0.611 \times 0.277 \tag{83}$$
$$= 0.169 \tag{84}$$

This numerical example illustrates how the LSTM cell processes a single input, selectively updates its internal memory (cell state), and produces an output (hidden state) based on its current memory and input.

LSTM Implementation for Time Series Anomaly Detection

LSTMs can be applied to time series anomaly detection through a predictive approach, where the network is trained to predict the next value in a sequence based on historical values. Significant deviations between predicted and actual values indicate potential anomalies. The following code demonstrates this approach:

Listing 5 LSTM for Time Series Anomaly Detection

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

def create_sequences(data, seq_length):
    """Create sequences for supervised learning"""
    X, y = [], []
    for i in range(len(data) - seq_length):
        X.append(data[i:i+seq_length])
        y.append(data[i+seq_length])
    return np.array(X), np.array(y)

def lstm_anomaly_detector(time_series, seq_length=20, threshold_percentile=95):
    """
    Detect anomalies using LSTM-based prediction.

    Parameters:
    -----------
    time_series : array-like
        Input time series data
    seq_length : int
        Length of input sequences
    threshold_percentile : float
```

```
        Percentile for anomaly threshold

    Returns:
    --------
    anomalies : array-like
        Boolean array indicating anomalies
    predictions : array-like
        LSTM predictions
    """
    # Scale data to [0, 1]
    scaler = MinMaxScaler(feature_range=(0, 1))
    ts_scaled = scaler.fit_transform(np.array(time_series).reshape(-1, 1)).flatten()

    # Create sequences
    X, y = create_sequences(ts_scaled, seq_length)
    X = X.reshape(X.shape[0], X.shape[1], 1)

    # Split data (80% train, 20% validation)
    X_train, X_val, y_train, y_val = train_test_split(
        X, y, test_size=0.2, shuffle=False)

    # Build LSTM model
    model = Sequential([
        LSTM(64, activation='relu', return_sequences=True, input_shape=(seq_length, 1)),
        Dropout(0.2),
        LSTM(32, activation='relu'),
        Dropout(0.2),
        Dense(16, activation='relu'),
        Dense(1)
    ])

    # Compile model
    model.compile(optimizer='adam', loss='mse')
    print(model.summary())

    # Train with early stopping
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=10,
        mode='min',
        restore_best_weights=True
    )

    history = model.fit(
        X_train, y_train,
        epochs=50,
        batch_size=32,
        validation_data=(X_val, y_val),
        callbacks=[early_stopping],
        verbose=1
    )

    # Predict on all sequences
    y_pred = model.predict(X).flatten()

    # Calculate prediction errors
    errors = np.abs(y - y_pred)

    # Set threshold based on error distribution
    threshold = np.percentile(errors, threshold_percentile)

    # Detect anomalies
    anomalies = errors > threshold

    # Map back to original time series length
    full_anomalies = np.zeros(len(time_series), dtype=bool)
```

```
    full_predictions = np.zeros(len(time_series)) * np.nan

    # First seq_length elements cannot be predicted with this approach
    full_anomalies[seq_length:] = anomalies

    # Denormalize predictions
    y_pred_denorm = scaler.inverse_transform(
        y_pred.reshape(-1, 1)).flatten()
    full_predictions[seq_length:] = y_pred_denorm

    # Visualize results
    plt.figure(figsize=(14, 7))

    plt.subplot(211)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Model Training History')
    plt.ylabel('Loss (MSE)')
    plt.xlabel('Epoch')
    plt.legend()

    plt.subplot(212)
    plt.plot(time_series, label='Original Data')
    plt.plot(full_predictions, 'r--', label='LSTM Predictions')
    plt.scatter(
        np.where(full_anomalies)[0],
        np.array(time_series)[full_anomalies],
        color='red', s=80, marker='X', label='Anomalies'
    )
    plt.title('LSTM Anomaly Detection')
    plt.legend()
    plt.tight_layout()
    plt.show()

    # Plot error distribution
    plt.figure(figsize=(10, 6))
    plt.hist(errors, bins=50, alpha=0.75)
    plt.axvline(threshold, color='red', linestyle='--',
                label=f'Threshold ({threshold_percentile}th percentile)')
    plt.title('Prediction Error Distribution')
    plt.xlabel('Absolute Error')
    plt.ylabel('Frequency')
    plt.legend()
    plt.show()

    return full_anomalies, full_predictions, threshold

# Example usage
np.random.seed(42)
t = np.arange(0, 1000)
# Generate time series with multiple seasonal components and trend
trend = 0.01 * t
daily = 10 * np.sin(2 * np.pi * t / 50)  # "Daily" pattern
weekly = 5 * np.sin(2 * np.pi * t / 250)  # "Weekly" pattern
noise = np.random.normal(0, 1, 1000)
ts = trend + daily + weekly + noise

# Insert anomalies
ts[350:355] += 15  # Collective anomaly
ts[700] += 20      # Point anomaly
ts[800:820] -= 10  # Extended anomaly

# Detect anomalies
anomalies, predictions, threshold = lstm_anomaly_detector(ts, seq_length=50)
print(f"Detected {sum(anomalies)} anomalies using threshold {threshold:.4f}")
```

Fig. 2 LSTM autoencoder architecture for anomaly detection, showing encoder, bottleneck, and decoder components.

This implementation demonstrates how LSTMs can effectively learn complex temporal patterns in time series data and identify observations that deviate from these patterns as potential anomalies. The model architecture includes stacked LSTM layers with dropout for regularization, capturing both short-term and long-term dependencies in the data.

1.4 LSTM Autoencoders for Anomaly Detection

While predictive approaches using LSTMs are effective, LSTM autoencoders represent a more sophisticated anomaly detection architecture that combines the sequential processing capabilities of LSTMs with the powerful unsupervised learning framework of autoencoders. This approach is particularly valuable when labeled anomaly data is scarce, as it allows the model to learn normal patterns solely from normal training data.

1.4.1 Autoencoder Architecture and Principle

An autoencoder is a neural network architecture trained to reconstruct its own input, consisting of an encoder that compresses the input into a lower-dimensional latent representation and a decoder that attempts to reconstruct the original input from this compressed representation. By constraining the network to pass information through a bottleneck (the latent space), it learns to capture the most salient features of the data.

In an LSTM autoencoder, both the encoder and decoder are implemented using LSTM layers. The encoder processes the input sequence and produces either the final hidden state or a sequence of hidden states. This compressed representation is then passed to the decoder, which attempts to reconstruct the original sequence.

The key insight for anomaly detection is that when trained exclusively on normal data, the autoencoder learns to efficiently reconstruct normal patterns. When presented with anomalous data containing unfamiliar patterns, the reconstruction quality degrades, resulting in higher reconstruction error. This error serves as a natural anomaly score.

1.4.2 Mathematical Formulation

Formally, given an input sequence $X = [x_1, x_2, \ldots, x_T]$ where each $x_t \in \mathbb{R}^d$, the LSTM encoder maps this to a latent representation z:

$$z = f_{\text{encoder}}(X) = \text{LSTM}_{\text{enc}}([x_1, x_2, \ldots, x_T]) \tag{85}$$

The decoder then attempts to reconstruct the original sequence from this representation:

$$\hat{X} = [\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_T] = f_{\text{decoder}}(z) = \text{LSTM}_{\text{dec}}(z) \tag{86}$$

The reconstruction error is calculated using a distance metric such as Mean Squared Error (MSE):

$$E(X) = \frac{1}{T}\sum_{t=1}^{T} \|x_t - \hat{x}_t\|^2 \tag{87}$$

For anomaly detection, we define a threshold τ based on the distribution of reconstruction errors on a validation set of normal data. A new sequence X_{new} is classified as anomalous if its reconstruction error exceeds this threshold:

$$\text{Anomaly}(X_{\text{new}}) = \begin{cases} 1, & \text{if } E(X_{\text{new}}) > \tau \\ 0, & \text{otherwise} \end{cases} \tag{88}$$

1.4.3 LSTM Autoencoder Implementation

The following code implements an LSTM autoencoder for time series anomaly detection:

Listing 6 LSTM Autoencoder for Anomaly Detection

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, LSTM, RepeatVector, TimeDistributed,
    Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

def create_sequences(data, seq_length):
    """Create overlapping sequences from time series data"""
    sequences = []
    for i in range(len(data) - seq_length + 1):
        sequences.append(data[i:i+seq_length])
    return np.array(sequences)

def lstm_autoencoder_anomaly_detector(time_series, seq_length=20,
                                      threshold_method='std', threshold_param=3):
    """
    Detect anomalies using LSTM Autoencoder.

    Parameters:
    -----------
    time_series : array-like
        Input time series data
    seq_length : int
        Length of input sequences
    threshold_method : str
        Method for threshold calculation: 'std' or 'percentile'
    threshold_param : float
        Parameter for threshold calculation:
        - If threshold_method='std', this is the number of standard deviations
        - If threshold_method='percentile', this is the percentile value

    Returns:
    --------
    anomalies : array-like
        Boolean array indicating anomalies (for each original time point)
    anomaly_scores : array-like
        Reconstruction error for each time point
```

```
    """
    # Scale data to [0, 1]
    scaler = MinMaxScaler(feature_range=(0, 1))
    ts_scaled = scaler.fit_transform(np.array(time_series).reshape(-1, 1)).flatten
        ()

    # Create sequences
    sequences = create_sequences(ts_scaled, seq_length)

    # Reshape for LSTM [samples, timesteps, features]
    sequences = sequences.reshape(sequences.shape[0], sequences.shape[1], 1)

    # Split data (70% train, 15% validation, 15% test)
    train_size = int(0.7 * len(sequences))
    val_size = int(0.15 * len(sequences))

    X_train = sequences[:train_size]
    X_val = sequences[train_size:train_size+val_size]
    X_test = sequences[train_size+val_size:]

    # Define model architecture
    # Input sequence
    inputs = Input(shape=(seq_length, 1))

    # Encoder
    x = LSTM(64, activation='relu', return_sequences=True)(inputs)
    x = Dropout(0.2)(x)
    x = LSTM(32, activation='relu', return_sequences=False)(x)
    x = Dropout(0.2)(x)

    # Bottleneck representation
    encoded = Dense(16)(x)

    # Decoder (prepare for LSTM by repeating the latent vector)
    x = RepeatVector(seq_length)(encoded)

    # Decoder LSTM layers
    x = LSTM(32, activation='relu', return_sequences=True)(x)
    x = Dropout(0.2)(x)
    x = LSTM(64, activation='relu', return_sequences=True)(x)
    x = Dropout(0.2)(x)

    # Output layer
    outputs = TimeDistributed(Dense(1))(x)

    # Build and compile model
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer='adam', loss='mse')

    print(model.summary())

    # Callbacks
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=10,
        mode='min',
        restore_best_weights=True
    )

    reduce_lr = ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,
        patience=5,
        min_lr=0.0001
    )

    # Train model on normal data only
    history = model.fit(
```

```
        X_train, X_train,  # Autoencoder reconstructs its own input
        epochs=50,
        batch_size=32,
        validation_data=(X_val, X_val),
        callbacks=[early_stopping, reduce_lr],
        verbose=1
    )

    # Predict reconstructions
    train_pred = model.predict(X_train)
    val_pred = model.predict(X_val)
    test_pred = model.predict(X_test)

    # Calculate reconstruction errors (MSE) for each sequence
    train_error = np.mean(np.square(X_train - train_pred), axis=(1, 2))
    val_error = np.mean(np.square(X_val - val_pred), axis=(1, 2))
    test_error = np.mean(np.square(X_test - test_pred), axis=(1, 2))

    # Calculate threshold from validation errors
    if threshold_method == 'std':
        # Mean + threshold_param * standard deviation
        error_mean = np.mean(val_error)
        error_std = np.std(val_error)
        threshold = error_mean + threshold_param * error_std
    else:  # 'percentile'
        # Use specified percentile of validation errors
        threshold = np.percentile(val_error, threshold_param)

    # Calculate reconstruction errors for full dataset
    all_sequences = create_sequences(ts_scaled, seq_length).reshape(-1, seq_length,
        1)
    all_pred = model.predict(all_sequences)
    all_error = np.mean(np.square(all_sequences - all_pred), axis=(1, 2))

    # Detect anomalies
    anomaly_sequences = all_error > threshold

    # Map sequence anomalies back to original points
    # A point is anomalous if it's part of an anomalous sequence
    point_anomaly_scores = np.zeros(len(ts_scaled))
    anomaly_count = np.zeros(len(ts_scaled))

    for i in range(len(anomaly_sequences)):
        # Each point gets the maximum score from any sequence it's part of
        for j in range(seq_length):
            idx = i + j
            if idx < len(point_anomaly_scores):
                point_anomaly_scores[idx] = max(point_anomaly_scores[idx],
                    all_error[i])
                anomaly_count[idx] += 1

    # Normalize by the number of sequences each point appears in
    for i in range(len(point_anomaly_scores)):
        if anomaly_count[i] > 0:
            point_anomaly_scores[i] /= anomaly_count[i]

    # Find anomalies in the original time series
    anomalies = point_anomaly_scores > threshold

    # Visualize results
    plt.figure(figsize=(15, 10))

    # Training history
    plt.subplot(311)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Model Training History')
    plt.ylabel('Loss (MSE)')
```

```
    plt.xlabel('Epoch')
    plt.legend()

    # Reconstruction error distribution
    plt.subplot(312)
    plt.hist(val_error, bins=50, alpha=0.5, label='Validation Errors')
    plt.hist(test_error, bins=50, alpha=0.5, label='Test Errors')
    plt.axvline(x=threshold, color='r', linestyle='--',
                label=f'Threshold: {threshold:.4f}')
    plt.title('Reconstruction Error Distribution')
    plt.xlabel('Mean Squared Error')
    plt.ylabel('Frequency')
    plt.legend()

    # Original data with anomalies
    plt.subplot(313)
    plt.plot(time_series, label='Original Data')
    plt.scatter(
        np.where(anomalies)[0],
        np.array(time_series)[anomalies],
        color='red', s=50, marker='X', label='Detected Anomalies'
    )
    # Plot reconstruction error as area in background
    ax2 = plt.gca().twinx()
    ax2.fill_between(
        range(len(point_anomaly_scores)),
        point_anomaly_scores,
        alpha=0.3,
        color='orange',
        label='Anomaly Score'
    )
    ax2.set_ylabel('Reconstruction Error')
    ax2.axhline(y=threshold, color='r', linestyle='--', alpha=0.3)

    plt.title('LSTM Autoencoder Anomaly Detection')
    plt.legend()
    plt.tight_layout()
    plt.show()

    # Example of a normal vs anomalous sequence
    plt.figure(figsize=(15, 6))

    # Find indices of normal and anomalous sequences
    normal_idx = np.where(all_error < threshold)[0][0]
    anomaly_idx = np.where(all_error > threshold)[0][0] if np.any(all_error >
         threshold) else 0

    plt.subplot(121)
    plt.plot(all_sequences[normal_idx, :, 0], label='Original')
    plt.plot(all_pred[normal_idx, :, 0], label='Reconstructed')
    plt.title(f'Normal Sequence (Error: {all_error[normal_idx]:.4f})')
    plt.legend()

    plt.subplot(122)
    plt.plot(all_sequences[anomaly_idx, :, 0], label='Original')
    plt.plot(all_pred[anomaly_idx, :, 0], label='Reconstructed')
    plt.title(f'Anomalous Sequence (Error: {all_error[anomaly_idx]:.4f})')
    plt.legend()

    plt.tight_layout()
    plt.show()

    return anomalies, point_anomaly_scores, threshold

# Example usage with a complex time series
np.random.seed(42)
t = np.arange(0, 1000)
# Generate time series with multiple seasonal components and trend
```

```
trend = 0.01 * t
daily = 10 * np.sin(2 * np.pi * t / 50)  # "Daily" pattern
weekly = 5 * np.sin(2 * np.pi * t / 250)  # "Weekly" pattern
noise = np.random.normal(0, 1, 1000)
ts = trend + daily + weekly + noise

# Insert anomalies
ts[350:355] += 15  # Collective anomaly
ts[700] += 20      # Point anomaly
ts[800:820] -= 10  # Extended anomaly

# Detect anomalies
anomalies, anomaly_scores, threshold = lstm_autoencoder_anomaly_detector(
    ts, seq_length=30, threshold_method='std', threshold_param=2.5)
print(f"Detected {sum(anomalies)} anomalies using threshold {threshold:.4f}")
```

This implementation demonstrates several advanced concepts for LSTM autoencoder-based anomaly detection:

1. Overlapping sequence creation to maximize data utilization 2. A comprehensive encoder-decoder architecture with regularization (dropout) 3. Sophisticated threshold calculation methods 4. Mapping sequence-level anomaly scores back to point-level scores 5. Visualization of normal vs. anomalous reconstructions

1.4.4 Numerical Example: Server CPU Utilization Monitoring

To illustrate how an LSTM autoencoder performs anomaly detection in practice, consider a simplified numerical example of server CPU utilization monitoring:

1. **Normal Sequence Processing:**
   - Input sequence (normal): $X = [10\%, 12\%, 11\%, 13\%, 12\%]$ (normalized to $[0.1, 0.12, 0.11, 0.13, 0.12]$)
   - The encoder compresses this to a latent vector (simplified): $z = [0.3, 0.7]$
   - The decoder reconstructs: $\hat{X} = [9.8\%, 12.3\%, 10.8\%, 13.2\%, 11.9\%]$ (or $[0.098, 0.123, 0.108, 0.132, 0.119]$)
   - Reconstruction error (MSE):

$$\text{MSE} = \frac{1}{5}\sum_{i=1}^{5}(x_i - \hat{x}_i)^2 \tag{89}$$

$$= \frac{1}{5}[(0.1 - 0.098)^2 + (0.12 - 0.123)^2 + (0.11 - 0.108)^2 + (0.13 - 0.132)^2 + (0.12 - 0.119)^2] \tag{90}$$

$$= \frac{1}{5}[0.000004 + 0.000009 + 0.000004 + 0.000004 + 0.000001] \tag{91}$$

$$= \frac{0.000022}{5} = 0.0000044 \tag{92}$$

   - This very low error indicates the sequence follows normal patterns learned by the model.

2. **Anomalous Sequence Processing:**

- Input sequence (anomalous): $X_{anom} = [10\%, 12\%, 50\%, 13\%, 12\%]$ (normalized to $[0.1, 0.12, 0.5, 0.13, 0.12]$)
- The encoder produces a different latent vector: $z_{anom} = [0.8, 0.4]$ (different due to anomaly)
- The decoder attempts reconstruction: $\hat{X}_{anom} = [10.2\%, 12.5\%, 21.0\%, 13.5\%, 12.2\%]$ (or $[0.102, 0.125, 0.21, 0.135, 0.122]$)
- Notice that the decoder "smooths out" the spike at position 3 (reconstructing 21% instead of 50%)
- Reconstruction error (MSE):

$$\text{MSE} = \frac{1}{5}[(0.1 - 0.102)^2 + (0.12 - 0.125)^2 + (0.5 - 0.21)^2 + (0.13 - 0.135)^2 + (0.12 - 0.122)^2] \tag{93}$$

$$= \frac{1}{5}[0.000004 + 0.000025 + 0.0841 + 0.000025 + 0.000004] \tag{94}$$

$$= \frac{0.084158}{5} = 0.0168 \tag{95}$$

- This error is approximately 3,800 times larger than the error for the normal sequence.

3. **Anomaly Decision:**
   - If the threshold based on normal validation data is, for example, $\mu + 3\sigma = 0.0000044 + 3 \times 0.00001 = 0.0000344$
   - Since $0.0168 \gg 0.0000344$, the sequence is confidently flagged as anomalous
   - Note how the largest contribution to the error (0.0841) comes precisely from the anomalous time step

This example demonstrates why LSTM autoencoders are effective for anomaly detection-they learn to reconstruct normal patterns with high fidelity, but struggle to reproduce anomalous patterns not encountered during training. The reconstruction error naturally serves as an anomaly score, with higher errors indicating greater deviation from learned normal patterns.

1.4.5 End-to-End Process for LSTM Autoencoder-Based Anomaly Detection

Implementing LSTM autoencoder-based anomaly detection in practice involves several key stages:

Data Preparation

1. **Cleaning:** Handle missing values and remove outliers from training data
2. **Normalization:** Scale features to similar ranges (e.g., 0-1 or z-score standardization)
3. **Sequence Generation:** Create overlapping windows from the time series

$$\text{For series} \{x_1, x_2, ..., x_N\} \text{ and window length } L \tag{96}$$

$$\text{Create sequences}\{[x_1, ..., x_L], [x_2, ..., x_{L+1}], ..., [x_{N-L+1}, ..., x_N]\} \tag{97}$$

Training Phase

1. Train the LSTM autoencoder on sequences from normal data only
2. Monitor validation loss to prevent overfitting
3. Save model weights at the lowest validation loss

Threshold Selection

1. Apply the trained model to a held-out validation set of normal data
2. Calculate reconstruction errors for each validation sequence
3. Set the anomaly threshold based on these errors:

$$\text{Statistical threshold} : \tau = \mu + k\sigma \text{ (e.g., } k = 3 \text{ for 99.7\% confidence)} \tag{98}$$

$$\text{Percentile threshold} : \tau = \text{percentile}_q(\text{errors}) \text{ (e.g., } q = 99) \tag{99}$$

Detection Phase

1. Preprocess new data identically to training data
2. Generate sequences and pass through the autoencoder
3. Calculate reconstruction error for each sequence
4. Flag as anomaly if error ¿ threshold

Post-Processing

1. Group contiguous anomalous sequences (anomaly event detection)
2. Filter out very short anomalies (noise reduction)
3. Calculate anomaly severity scores based on reconstruction error magnitude

1.4.6 Case Study: ECG Anomaly Detection

LSTM autoencoders have been successfully applied to medical time series analysis, particularly electrocardiogram (ECG) data for detecting cardiac anomalies. The following case study illustrates this application:

Problem Statement

ECG signals contain subtle patterns indicating cardiac conditions. Manual review is time-consuming and error-prone, creating a need for automated detection of irregular heartbeat patterns.

Dataset

The MIT-BIH Arrhythmia Database containing:

- 48 half-hour ECG recordings (approximately 24 hours of cardiac data)
- Over 109,000 annotated beats
- Multiple arrhythmia types (important for comprehensive evaluation)

Approach

- Extract 5-second ECG segments at 100Hz (500 samples per segment)
- Train LSTM autoencoder exclusively on normal sinus rhythm segments
- Test on both normal and arrhythmic segments

Model Architecture

- Encoder: LSTM(128) $\rightarrow$ LSTM(64) $\rightarrow$ LSTM(32)
- Bottleneck: 16 dimensions
- Decoder: LSTM(32) $\rightarrow$ LSTM(64) $\rightarrow$ LSTM(128) $\rightarrow$ Dense(1)

Results

- Normal segments: Mean reconstruction error = $0.037(\sigma = 0.011)$
- Arrhythmic segments: Mean reconstruction error = $0.218(\sigma = 0.083)$
- Performance metrics: $ROC - AUC = 0.967, Precision = 0.93, Recall = 0.91$
- Particularly effective for ventricular arrhythmias (most dangerous type)

Clinical Impact

- 5$\times$ faster preliminary screening than manual review
- 71% reduction in false alarms compared to rule-based systems
- Localized anomalies within long recordings

This case study demonstrates the practical application of LSTM autoencoders in healthcare for detecting anomalies in physiological time series, with significant potential for clinical decision support.

2 Graph-Based Anomaly Detection

While time series represent sequential data with temporal dependencies, many real-world systems are better modeled as networks or graphs, where entities and their relationships form complex structures. Graph-based anomaly detection focuses on identifying unusual patterns within these relational structures, complementing time-based methods for comprehensive anomaly detection across different data representations.

2.1 Graph Fundamentals

A graph $G = (V, E)$ consists of a set of vertices (nodes) V and a set of edges E that represent relationships between pairs of vertices. Graphs provide a natural representation for numerous real-world systems including social networks (people connected by friendships or interactions), financial transaction networks, telecommunications networks, protein-protein interaction networks, and web page linkage structures.

2.1.1 Graph Types and Representations

Graphs can be categorized along several dimensions:

Directed vs. Undirected Graphs

- **Undirected Graphs:** Edges represent symmetric relationships (e.g., friendship on Facebook). An edge between nodes i and j is represented as an unordered pair $\{i, j\}$.
- **Directed Graphs:** Edges have orientation, representing asymmetric relationships (e.g., following on Twitter). An edge from node i to node j is represented as an ordered pair (i, j).

Weighted vs. Unweighted Graphs

- **Unweighted Graphs:** Edges simply indicate the presence of a relationship.
- **Weighted Graphs:** Edges carry values indicating relationship strength or distance (e.g., transaction amounts in financial networks, communication frequency in social networks).

Simple vs. Complex Graphs

- **Simple Graphs:** No self-loops (edges from a node to itself) or multiple edges between the same pair of nodes.
- **Multigraphs:** Allow multiple edges between the same pair of nodes.
- **Hypergraphs:** Edges can connect more than two nodes simultaneously.

Attributed Graphs

In many applications, both nodes and edges have associated attributes or features. An attributed graph is formally represented as $G = (V, E, X_V, X_E)$, where:

- X_V is a matrix of node features, where $X_V[i]$ represents the feature vector of node i
- X_E is a matrix of edge features, where $X_E[i, j]$ represents the feature vector of edge (i, j)

2.1.2 Graph Representations for Computation

Several mathematical representations are used to work with graphs computationally:

Adjacency Matrix

The adjacency matrix $A \in \{0, 1\}^{n \times n}$ for an unweighted graph with n nodes has entries:

$$A_{ij} = \begin{cases} 1 & \text{if edge } (i, j) \text{ exists} \\ 0 & \text{otherwise} \end{cases} \tag{100}$$

For weighted graphs, A_{ij} contains the weight of edge (i, j). For undirected graphs, A is symmetric ($A_{ij} = A_{ji}$).

Degree Matrix

The degree matrix D is a diagonal matrix where D_{ii} is the degree (number of connections) of node i:

$$D_{ii} = \sum_j A_{ij} \tag{101}$$

Laplacian Matrix

The Laplacian matrix $L = D - A$ captures both connectivity and degree information. It has special spectral properties useful for graph partitioning and clustering.

Node Feature Matrix

The node feature matrix $X \in \mathbb{R}^{n \times d}$ contains the d-dimensional feature vectors for each of the n nodes.

Edge List

An edge list representation stores each edge as a tuple $(i, j, [w_{ij}])$, where i and j are node indices and w_{ij} is the optional edge weight.

2.1.3 Example Graph Construction

The following code demonstrates the construction and visualization of a simple attributed graph:

Listing 7 Graph Construction and Visualization

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

def create_sample_graph(n_nodes=20, edge_probability=0.2, seed=42):
    """
        Create a sample attributed graph for demonstration.

    Parameters:
    -----------
    n_nodes : int
        Number of nodes in the graph
    edge_probability : float
        Probability of edge creation between any two nodes
    seed : int
        Random seed for reproducibility

    Returns:
    --------
    G : networkx.Graph
        Graph with node features and edge weights
    """
    np.random.seed(seed)

    # Create a random graph
    G = nx.erdos_renyi_graph(n=n_nodes, p=edge_probability, seed=seed)

    # Add node features (2-dimensional for visualization simplicity)
    for i in range(n_nodes):
        # Generate random features
        features = np.random.normal(size=2)
        G.nodes[i]['features'] = features

    # Add edge weights
    for u, v in G.edges():
        # Weight based on feature similarity and random component
        f_u = G.nodes[u]['features']
        f_v = G.nodes[v]['features']
        similarity = 1 / (1 + np.linalg.norm(f_u - f_v))  # Closer features ->
            higher weights
```

```
        random_component = np.random.uniform(0.5, 1.5)
        G[u][v]['weight'] = similarity * random_component

    # Add anomalous nodes with unusual features or connections
    # Anomaly 1: Node with unusual features
    anomaly_node1 = n_nodes - 2
    G.nodes[anomaly_node1]['features'] = np.array([5.0, 5.0])  # Outlier features
    G.nodes[anomaly_node1]['anomaly_type'] = 'feature'

    # Anomaly 2: Node with unusually high connectivity
    anomaly_node2 = n_nodes - 1
    for i in range(int(n_nodes/2)):
        if i != anomaly_node2 and not G.has_edge(i, anomaly_node2):
            G.add_edge(i, anomaly_node2, weight=np.random.uniform(0.1, 0.3))
    G.nodes[anomaly_node2]['anomaly_type'] = 'structural'

    return G

def visualize_graph(G, with_features=False):
    """
    Visualize a graph with optional feature visualization.

    Parameters:
    -----------
    G : networkx.Graph
        Graph to visualize
    with_features : bool
        Whether to show node features in a separate plot
    """
    plt.figure(figsize=(12, 6))

    # Node colors based on anomaly type
    node_colors = []
    for n in G.nodes():
        if 'anomaly_type' in G.nodes[n]:
            if G.nodes[n]['anomaly_type'] == 'feature':
                node_colors.append('red')
            elif G.nodes[n]['anomaly_type'] == 'structural':
                node_colors.append('orange')
            else:
                node_colors.append('blue')
        else:
            node_colors.append('blue')

    # Node sizes based on degree
    node_sizes = [300 + 100 * G.degree(n) for n in G.nodes()]

    # Edge weights
    edge_weights = [G[u][v]['weight'] * 2 for u, v in G.edges()]

    # Graph visualization
    plt.subplot(1, 2 if with_features else 1, 1)
    pos = nx.spring_layout(G, seed=42)  # Position nodes using Fruchterman-Reingold
        force-directed algorithm
    nx.draw(G, pos, with_labels=True, node_color=node_colors, node_size=node_sizes,
            width=edge_weights, edge_color='gray', alpha=0.7)
    plt.title('Graph Structure')

    if with_features:
        # Feature visualization
        plt.subplot(1, 2, 2)
        features = np.array([G.nodes[n]['features'] for n in G.nodes()])

        # Scatter plot of node features
        plt.scatter(features[:, 0], features[:, 1], c=node_colors, s=node_sizes,
            alpha=0.7)

        # Add node labels
```

```
        for i, (x, y) in enumerate(features):
            plt.text(x, y, str(i), fontsize=9)

        plt.title('Node Features')
        plt.xlabel('Feature 1')
        plt.ylabel('Feature 2')
        plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

# Create and visualize a sample graph
sample_graph = create_sample_graph(n_nodes=15, edge_probability=0.2)
visualize_graph(sample_graph, with_features=True)

# Extract adjacency and feature matrices
def graph_to_matrices(G):
    """Convert a graph to adjacency and feature matrices"""
    # Get number of nodes
    n_nodes = G.number_of_nodes()

    # Create adjacency matrix
    A = nx.to_numpy_array(G)

    # Create feature matrix
    X = np.zeros((n_nodes, 2))
    for i in range(n_nodes):
        X[i] = G.nodes[i]['features']

    return A, X

A, X = graph_to_matrices(sample_graph)
print("Adjacency Matrix:")
print(A)
print("\nFeature Matrix:")
print(X)
```

These code examples demonstrate the construction of a sample graph with node features and edge weights, visualization of the graph structure and node features, and extraction of the graph's adjacency and feature matrices for further analysis. The sample graph includes deliberately injected anomalies: a node with unusual feature values and a node with unusually high connectivity, which will serve as ground truth for evaluating anomaly detection methods.

2.2 Traditional Graph Anomaly Detection Methods

Before delving into advanced graph representation learning techniques, we first explore traditional graph anomaly detection approaches that rely on statistical analysis of graph structure. These methods have the advantage of interpretability and often serve as baselines for more sophisticated techniques.

2.2.1 Types of Graph Anomalies

Graph anomalies can manifest in several forms:

Node Anomalies

- **Structural Node Anomalies:** Nodes with unusual connectivity patterns (e.g., extremely high/low degree, unusual clustering coefficient)

- **Feature-based Node Anomalies:** Nodes with attribute values that deviate significantly from the norm
- **Contextual Node Anomalies:** Nodes whose features are inconsistent with their structural neighborhood

Edge Anomalies

- **Unexpected Connections:** Edges between nodes that should not be related based on their features or communities
- **Weight Anomalies:** Edges with weights that are unusually high or low given the connected nodes
- **Temporal Edge Anomalies:** Sudden appearance or disappearance of edges in dynamic graphs

Subgraph Anomalies

- **Density Anomalies:** Unusually dense or sparse regions in an otherwise uniform graph
- **Structure Anomalies:** Subgraphs with unusual topological patterns (e.g., near-cliques, stars, chains)
- **Community Anomalies:** Groups of nodes that form unexpected community structures

2.2.2 Statistical Approaches

Statistical approaches to graph anomaly detection analyze various graph metrics to identify nodes, edges, or subgraphs that deviate significantly from expected patterns.

Degree-based Detection

One of the simplest approaches examines the degree distribution of the graph. For a node v with degree d_v in a graph with mean degree μ_d and standard deviation σ_d, we can compute a Z-score:

$$Z(v) = \frac{d_v - \mu_d}{\sigma_d} \tag{102}$$

Nodes with $|Z(v)| > k$ (typically $k = 3$) can be flagged as anomalies. This method effectively identifies unusually connected nodes but fails to capture more subtle structural anomalies.

Centrality Measures

Various centrality measures quantify the importance of nodes in different ways:

- **Betweenness Centrality:** Measures how often a node appears on shortest paths between other nodes:
$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{103}$$
where σ_{st} is the number of shortest paths from node s to node t, and $\sigma_{st}(v)$ is the number of those paths passing through v.

- **Closeness Centrality:** Measures how close a node is to all other nodes:

$$CC(v) = \frac{n-1}{\sum_{u \neq v} d(v,u)} \tag{104}$$

  where $d(v,u)$ is the shortest path distance between nodes v and u, and n is the number of nodes.
- **Eigenvector Centrality:** Measures node importance based on the importance of its neighbors:

$$EC(v) = \frac{1}{\lambda} \sum_{u \in \mathcal{N}(v)} A_{vu} \cdot EC(u) \tag{105}$$

  where λ is a constant (the eigenvalue) and $\mathcal{N}(v)$ is the set of neighbors of v.

Anomalies can be detected by identifying nodes with unusually high or low centrality values relative to the distribution of the specific centrality measure across all nodes.

Local Structure Analysis

Local structural measures focus on patterns within a node's immediate neighborhood:

- **Clustering Coefficient:** Measures the degree to which a node's neighbors are connected to each other:

$$C_v = \frac{2|\{e_{jk} : v_j, v_k \in \mathcal{N}(v), e_{jk} \in E\}|}{|\mathcal{N}(v)|(|\mathcal{N}(v)| - 1)} \tag{106}$$

  where $\mathcal{N}(v)$ is the neighborhood of node v.
- **Local Outlier Factor (LOF):** Adapted for graphs by using structural features, LOF compares the local density of a node to the local densities of its neighbors.
- **Neighborhood Pattern Anomaly:** Compares the distribution of connections in a node's ego network (the subgraph formed by the node and its neighbors) with expected patterns.

2.2.3 Implementation of Statistical Graph Anomaly Detection

The following code demonstrates several statistical approaches to graph anomaly detection:

Listing 8 Statistical Graph Anomaly Detection Methods

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy import stats

def statistical_graph_anomaly_detection(G, k=2):
    """
    Detect anomalies in a graph using statistical methods.

    Parameters:
    -----------
```

```
    G : networkx.Graph
        Input graph
    k : float
        Threshold multiplier for z-scores

    Returns:
    --------
    results : dict
        Dictionary of anomaly scores and detected anomalies
    """
    n = G.number_of_nodes()
    results = {
        "degree_z_scores": {},
        "clustering_z_scores": {},
        "betweenness_z_scores": {},
        "combined_scores": {},
        "anomalies": set()
    }

    # Compute graph metrics
    degrees = [d for _, d in G.degree()]
    clustering_coeffs = list(nx.clustering(G).values())
    betweenness = list(nx.betweenness_centrality(G).values())

    # Calculate statistics
    degree_mean, degree_std = np.mean(degrees), np.std(degrees)
    clustering_mean, clustering_std = np.mean(clustering_coeffs), np.std(
        clustering_coeffs)
    betweenness_mean, betweenness_std = np.mean(betweenness), np.std(betweenness)

    # Handle zero standard deviation edge case
    if degree_std == 0:
        degree_std = 1e-10
    if clustering_std == 0:
        clustering_std = 1e-10
    if betweenness_std == 0:
        betweenness_std = 1e-10

    # Calculate z-scores for each node
    for node in G.nodes():
        # Degree z-score
        degree = G.degree(node)
        degree_z = (degree - degree_mean) / degree_std
        results["degree_z_scores"][node] = degree_z

        # Clustering coefficient z-score
        clustering = nx.clustering(G, node)
        clustering_z = (clustering - clustering_mean) / clustering_std
        results["clustering_z_scores"][node] = clustering_z

        # Betweenness centrality z-score
        bc = nx.betweenness_centrality(G)[node]
        betweenness_z = (bc - betweenness_mean) / betweenness_std
        results["betweenness_z_scores"][node] = betweenness_z

        # Combined anomaly score (max absolute z-score across metrics)
        combined_score = max(abs(degree_z), abs(clustering_z), abs(betweenness_z))
        results["combined_scores"][node] = combined_score

        # Detect anomalies: if any z-score exceeds threshold k
        if abs(degree_z) > k or abs(clustering_z) > k or abs(betweenness_z) > k:
            results["anomalies"].add(node)

    return results

def visualize_anomaly_results(G, results):
    """
    Visualize the graph with detected anomalies and metrics.
```

```
80
81      Parameters:
82      -----------
83      G : networkx.Graph
84          Input graph
85      results : dict
86          Results from statistical_graph_anomaly_detection
87      """
88      # Prepare node colors based on anomaly status
89      node_colors = ['red' if node in results["anomalies"] else 'skyblue' for node in
            G.nodes()]
90
91      # Node sizes based on combined score
92      node_sizes = [100 + 500 * abs(results["combined_scores"][node]) for node in G.
            nodes()]
93
94      # Create metrics dataframe
95      metrics_df = pd.DataFrame({
96          'Node': list(G.nodes()),
97          'Degree': [G.degree(node) for node in G.nodes()],
98          'Degree Z-score': [results["degree_z_scores"][node] for node in G.nodes()],
99          'Clustering Z-score': [results["clustering_z_scores"][node] for node in G.
                nodes()],
100         'Betweenness Z-score': [results["betweenness_z_scores"][node] for node in G
                .nodes()],
101         'Combined Score': [results["combined_scores"][node] for node in G.nodes()],
102         'Is Anomaly': [node in results["anomalies"] for node in G.nodes()]
103     })
104
105     # Sort by combined score
106     metrics_df = metrics_df.sort_values('Combined Score', ascending=False)
107
108     # Visualize
109     plt.figure(figsize=(18, 10))
110
111     # Graph visualization
112     plt.subplot(2, 2, 1)
113     pos = nx.spring_layout(G, seed=42)
114     nx.draw(G, pos, with_labels=True, node_color=node_colors, node_size=node_sizes,
115             edge_color='gray', alpha=0.7)
116     plt.title('Graph with Detected Anomalies')
117
118     # Z-score distribution plots
119     plt.subplot(2, 2, 2)
120     plt.hist(
121         [metrics_df["Degree Z-score"], metrics_df["Clustering Z-score"], metrics_df
                ["Betweenness Z-score"]],
122         bins=20, alpha=0.7,
123         label=['Degree', 'Clustering', 'Betweenness']
124     )
125     plt.axvline(x=0, color='k', linestyle='--', alpha=0.3)
126     plt.title('Z-score Distributions')
127     plt.legend()
128     plt.grid(True, alpha=0.3)
129
130     # Top anomalies table
131     plt.subplot(2, 2, 3)
132     plt.axis('off')
133     top_anomalies = metrics_df.head(10)
134     table_text = []
135
136     # Create table content
137     table_text.append(['Node', 'Degree Z', 'Clust Z', 'Betw Z', 'Combined'])
138     for _, row in top_anomalies.iterrows():
139         table_text.append([
140             str(int(row['Node'])),
141             f"{row['Degree Z-score']:.2f}",
142             f"{row['Clustering Z-score']:.2f}",
```

```
            f"{row['Betweenness Z-score']:.2f}",
            f"{row['Combined Score']:.2f}"
        ])

    # Create table
    plt.table(
        cellText=table_text,
        loc='center',
        cellLoc='center',
        colWidths=[0.15, 0.2, 0.2, 0.2, 0.2]
    )
    plt.title('Top Anomalies')

    # Feature plot if features exist
    plt.subplot(2, 2, 4)
    if 'features' in G.nodes[0]:
        features = np.array([G.nodes[n]['features'] for n in G.nodes()])
        plt.scatter(
            features[:, 0],
            features[:, 1],
            c=node_colors,
            s=node_sizes,
            alpha=0.7
        )

        # Add node labels
        for i, (x, y) in enumerate(features):
            plt.text(x, y, str(i), fontsize=9)

        plt.title('Node Features with Anomaly Scores')
        plt.xlabel('Feature 1')
        plt.ylabel('Feature 2')
        plt.grid(True, alpha=0.3)
    else:
        plt.text(0.5, 0.5, 'No node features available', horizontalalignment='
            center', verticalalignment='center')
        plt.title('Feature Space')

    plt.tight_layout()
    plt.show()

    return metrics_df

# Example usage with our sample graph
anomaly_results = statistical_graph_anomaly_detection(sample_graph, k=2)
metrics_df = visualize_anomaly_results(sample_graph, anomaly_results)
print("\nDetected Anomalies:", anomaly_results["anomalies"])
print("\nTop 5 nodes by anomaly score:")
print(metrics_df[['Node', 'Combined Score', 'Is Anomaly']].head(5))
```

This implementation calculates multiple statistical measures (degree, clustering coefficient, and betweenness centrality), computes z-scores for each, and identifies anomalies based on these scores. The visualization shows the graph with highlighted anomalies, distributions of the statistical measures, and a table of the top anomalies.

2.2.4 Strengths and Limitations of Statistical Approaches

Statistical approaches to graph anomaly detection offer several advantages:

- **Interpretability:** Results are easily explainable (e.g., "Node X is anomalous because its degree is 5 standard deviations above the mean")
- **Computational Efficiency:** Many metrics can be calculated efficiently even for large graphs

- **No Training Required:** Most methods are unsupervised and require no labeled training data

However, they also have significant limitations:

- **Limited to Simple Patterns:** Struggle to capture complex, multi-dimensional anomalies
- **Feature-Structure Disconnect:** Often fail to incorporate both structural and attribute information effectively
- **Threshold Selection:** Choosing appropriate thresholds requires domain expertise and can be subjective
- **Local Focus:** Many methods analyze metrics in isolation, missing patterns that emerge from combinations of features

These limitations have motivated the development of more sophisticated approaches based on graph representation learning.

2.3 Graph Embeddings

Graph embeddings address many limitations of traditional statistical approaches by mapping graph elements (nodes, edges, or subgraphs) to continuous vector spaces. This enables application of standard machine learning techniques to graphs and captures complex patterns that statistical measures might miss.

2.3.1 Graph Embedding Fundamentals

A graph embedding is a function $f : V \to \mathbb{R}^d$ that assigns each node $v \in V$ to a d-dimensional vector $\mathbf{z}_v \in \mathbb{R}^d$, where typically $d \ll |V|$. The key desideratum is that the vector representation preserves graph properties of interest: nodes that are "similar" in the graph should have similar vector representations.

Different embedding methods define similarity differently:

- **First-order Proximity:** Directly connected nodes should have similar embeddings
- **Second-order Proximity:** Nodes with similar neighbors should have similar embeddings
- **Structural Equivalence:** Nodes with similar structural roles should have similar embeddings
- **Community Proximity:** Nodes in the same community should have similar embeddings

The objective function for learning embeddings typically balances two terms:

$$\mathcal{L} = \sum_{(u,v)\in E} \text{similarity}(\mathbf{z}_u, \mathbf{z}_v) - \sum_{(u,v)\notin E} \text{similarity}(\mathbf{z}_u, \mathbf{z}_v) \tag{107}$$

This encourages connected nodes to have similar embeddings while pushing unconnected nodes' embeddings apart.

2.3.2 Random Walk-Based Embeddings

Random walk-based embedding methods like DeepWalk and Node2Vec generate "sentences" of nodes by performing random walks on the graph, then apply techniques from natural language processing to learn node representations based on co-occurrence in these walks.

DeepWalk

DeepWalk, introduced by Perozzi et al. (2014), pioneered the application of word2vec-style embeddings to graphs. The algorithm:

1. Performs uniform random walks from each node, generating sequences of nodes
2. Treats these sequences as "sentences" where each node is a "word"
3. Applies the Skip-gram model from word2vec to learn embeddings

The Skip-gram objective maximizes the probability of observing context nodes around each center node:

$$\max_{\theta} \sum_{v \in V} \sum_{c \in N(v)} \log P(c|v;\theta) \tag{108}$$

where $N(v)$ is the set of neighboring nodes of v observed in random walks.

Node2Vec

Node2Vec, proposed by Grover and Leskovec (2016), extends DeepWalk by introducing biased random walks controlled by two parameters:

- **Return Parameter (p):** Controls the likelihood of returning to the previous node
- **In-out Parameter (q):** Controls whether the walk explores the broader network (BFS-like) or stays close to the starting node (DFS-like)

The transition probability from current node t to next node x (given previous node v) is:

$$P(x|t,v) = \begin{cases} \frac{\alpha_{pq}(t,x) \cdot w_{tx}}{Z} & \text{if } (t,x) \in E \\ 0 & \text{otherwise} \end{cases} \tag{109}$$

where w_{tx} is the edge weight, Z is a normalization constant, and:

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{vx} = 0 \text{ (return)} \\ 1 & \text{if } d_{vx} = 1 \text{ (one step away)} \\ \frac{1}{q} & \text{if } d_{vx} = 2 \text{ (further away)} \end{cases} \tag{110}$$

with d_{vx} being the shortest path distance between nodes v and x.
By tuning p and q, Node2Vec can focus on different types of similarities:

- p small, q large: Focus on structural equivalence
- p large, q small: Focus on homophily (community structure)

Implementation of Node2Vec for Anomaly Detection

The following code demonstrates implementing Node2Vec embeddings for graph anomaly detection:

Listing 9 Node2Vec for Graph Anomaly Detection

```
from node2vec import Node2Vec
from sklearn.manifold import TSNE
from sklearn.neighbors import LocalOutlierFactor
import matplotlib.pyplot as plt
import numpy as np
import networkx as nx

def node2vec_anomaly_detection(G, dimensions=16, walk_length=30, num_walks=200,
                               p=1.0, q=1.0, contamination=0.1):
    """
    Detect anomalies using Node2Vec embeddings and Local Outlier Factor.

    Parameters:
    -----------
    G : networkx.Graph
        Input graph
    dimensions : int
        Embedding dimensions
    walk_length : int
        Length of each random walk
    num_walks : int
        Number of random walks per node
    p : float
        Return parameter (lower values encourage returning to previous nodes)
    q : float
        In-out parameter (lower values encourage exploration, higher values keep walks localized)
    contamination : float
        Expected proportion of anomalies

    Returns:
    --------
    anomalies : set
        Set of anomalous node indices
    scores : dict
        Dictionary of anomaly scores for each node
    embeddings : dict
        Dictionary of node embeddings
    """
    # Generate walks and learn embeddings
    print("Generating Node2Vec embeddings...")
    node2vec = Node2Vec(
        G,
        dimensions=dimensions,
        walk_length=walk_length,
        num_walks=num_walks,
        p=p,
        q=q,
        workers=4
    )
    model = node2vec.fit(window=10, min_count=1)

    # Extract node embeddings
    embeddings = {node: model.wv[str(node)] for node in G.nodes()}
    embeddings_matrix = np.array([embeddings[node] for node in sorted(G.nodes())])

    # Apply Local Outlier Factor for anomaly detection
    lof = LocalOutlierFactor(n_neighbors=min(20, len(G.nodes()) - 1), contamination=contamination)
    lof.fit_predict(embeddings_matrix)

```

```
    # Extract negative outlier factors (higher is more anomalous)
    outlier_scores = -lof.negative_outlier_factor_

    # Sort nodes by anomaly score
    sorted_nodes = sorted(G.nodes(), key=lambda x: outlier_scores[x], reverse=True)
    num_anomalies = int(contamination * len(G.nodes()))
    anomalies = set(sorted_nodes[:num_anomalies])

    # Create scores dictionary
    scores = {node: outlier_scores[node] for node in G.nodes()}

    return anomalies, scores, embeddings

def visualize_embeddings(G, embeddings, anomalies=None):
    """
    Visualize node embeddings in 2D using t-SNE.

    Parameters:
    -----------
    G : networkx.Graph
        Input graph
    embeddings : dict
        Dictionary of node embeddings
    anomalies : set
        Set of anomalous nodes
    """
    # Convert embeddings to matrix
    nodes = sorted(G.nodes())
    embeddings_matrix = np.array([embeddings[node] for node in nodes])

    # Reduce dimensions with t-SNE
    tsne = TSNE(n_components=2, perplexity=min(30, len(nodes)-1), n_iter=300,
         random_state=42)
    embeddings_2d = tsne.fit_transform(embeddings_matrix)

    # Prepare node colors based on anomaly status
    if anomalies is None:
        anomalies = set()
    node_colors = ['red' if node in anomalies else 'skyblue' for node in nodes]

    # Visualize embeddings
    plt.figure(figsize=(12, 10))

    # Original graph visualization
    plt.subplot(2, 1, 1)
    pos = nx.spring_layout(G, seed=42)
    nx.draw(G, pos, node_color=node_colors, with_labels=True)
    plt.title('Original Graph (Red = Detected Anomalies)')

    # Embedding visualization
    plt.subplot(2, 1, 2)
    plt.scatter(embeddings_2d[:, 0], embeddings_2d[:, 1], c=node_colors)

    # Add node labels
    for i, (x, y) in enumerate(embeddings_2d):
        plt.text(x, y, str(nodes[i]), fontsize=9)

    plt.title('Node2Vec Embeddings Visualization (t-SNE)')
    plt.xlabel('t-SNE Component 1')
    plt.ylabel('t-SNE Component 2')
    plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

    return embeddings_2d

# Example usage
```

```python
def compare_node2vec_params(G):
    """Compare different Node2Vec parameter settings"""
    settings = [
        {'p': 0.5, 'q': 2.0, 'title': 'Structural Equivalence (p=0.5, q=2.0)'},
        {'p': 1.0, 'q': 1.0, 'title': 'Balanced (p=1.0, q=1.0)'},
        {'p': 2.0, 'q': 0.5, 'title': 'Homophily/Communities (p=2.0, q=0.5)'}
    ]

    plt.figure(figsize=(18, 6))

    for i, setting in enumerate(settings):
        # Generate embeddings with specific parameters
        anomalies, scores, embeddings = node2vec_anomaly_detection(
            G, dimensions=16, p=setting['p'], q=setting['q'])

        # Reduce dimensions
        nodes = sorted(G.nodes())
        embeddings_matrix = np.array([embeddings[node] for node in nodes])
        tsne = TSNE(n_components=2, perplexity=min(30, len(nodes)-1), random_state
            =42)
        embeddings_2d = tsne.fit_transform(embeddings_matrix)

        # Node colors based on anomaly score
        node_colors = [scores[node] for node in nodes]

        # Plot
        plt.subplot(1, 3, i+1)
        scatter = plt.scatter(
            embeddings_2d[:, 0],
            embeddings_2d[:, 1],
            c=node_colors,
            cmap='YlOrRd',
            s=100,
            alpha=0.8
        )

        # Add node labels
        for j, (x, y) in enumerate(embeddings_2d):
            plt.text(x, y, str(nodes[j]), fontsize=9)

        plt.title(setting['title'])
        plt.colorbar(scatter, label='Anomaly Score')
        plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

# Create a larger sample graph with ground truth anomalies
larger_sample = create_sample_graph(n_nodes=50, edge_probability=0.1, seed=42)

# Compare Node2Vec parameters
compare_node2vec_params(larger_sample)

# Detect anomalies using Node2Vec
anomalies, scores, embeddings = node2vec_anomaly_detection(
    larger_sample, dimensions=16, p=1.0, q=1.0)
print(f"Detected {len(anomalies)} anomalies: {anomalies}")

# Visualize embeddings
embeddings_2d = visualize_embeddings(larger_sample, embeddings, anomalies)
```

This implementation demonstrates how Node2Vec embeddings can be used for anomaly detection by (1) generating biased random walks on the graph, (2) training a skip-gram model to learn node embeddings, (3) applying an outlier detection algorithm (Local Outlier Factor) to the embeddings, and (4) visualizing both the original graph and the learned embeddings.

2.3.3 Other Graph Embedding Approaches

Beyond random walk-based methods, several other approaches generate effective graph embeddings:

Matrix Factorization-Based Methods

These approaches factorize graph-related matrices to obtain embeddings:

- **Graph Factorization:** Directly factorizes the adjacency matrix: $A \approx ZZ^T$
- **GraRep:** Factorizes higher-order proximity matrices
- **HOPE (High-Order Proximity preserved Embedding):** Factorizes similarity matrices that capture higher-order proximities

Deep Learning-Based Methods

These approaches use neural networks to learn embeddings:

- **SDNE (Structural Deep Network Embedding):** Uses autoencoders to capture both first and second-order proximities
- **DNGR (Deep Neural Networks for Graph Representations):** Combines random walks with deep autoencoders
- **GraphSAGE:** Learns embeddings by sampling and aggregating features from a node's local neighborhood

2.3.4 Graph Embeddings for Anomaly Detection

Once nodes are embedded in vector space, traditional anomaly detection algorithms can be applied to identify outliers in this space:

Distance-based Detection:

Compute the centroid of all embeddings and flag nodes whose embeddings are far from this centroid:

$$\text{score}(v) = \|\mathbf{z}_v - \frac{1}{|V|}\sum_{u \in V} \mathbf{z}_u\|^2 \tag{111}$$

Density-based Detection:

Apply density-based outlier detection algorithms like Local Outlier Factor (LOF) directly to the embeddings. LOF compares the local density of a point to the densities of its neighbors:

$$\text{LOF}_k(v) = \frac{\sum_{o \in N_k(v)} \frac{\text{lrd}_k(o)}{\text{lrd}_k(v)}}{|N_k(v)|} \tag{112}$$

where $\text{lrd}_k(v)$ is the local reachability density and $N_k(v)$ is the k-neighborhood of node v in the embedding space.

Isolation-based Detection:

Algorithms like Isolation Forest recursively partition the embedding space and identify points that require fewer partitions to isolate (anomalies).

Cluster-based Detection:

Apply clustering to the embeddings and flag nodes that are far from cluster centers or form small, isolated clusters.

2.4 Graph Neural Networks (GNNs)

Graph Neural Networks represent the state-of-the-art in graph representation learning. Unlike traditional embeddings, which are essentially lookup tables, GNNs compute embeddings as a function of node features and neighborhood structure, allowing them to generalize to unseen nodes and incorporate both structural and attribute information.

2.4.1 GNN Fundamentals

The core principle behind GNNs is message passing: nodes iteratively aggregate information from their neighbors to update their representations. After L layers (iterations), each node's representation captures information from its L-hop neighborhood.

A general message passing update for node v at layer $l+1$ is:

$$\mathbf{h}_v^{(l+1)} = \text{UPDATE}^{(l)}\left(\mathbf{h}_v^{(l)}, \text{AGGREGATE}^{(l)}\left(\{\mathbf{h}_u^{(l)} : u \in \mathcal{N}(v)\}\right)\right) \tag{113}$$

where:

- $\mathbf{h}_v^{(l)}$ is the representation of node v at layer l
- $\mathcal{N}(v)$ is the set of neighbors of node v
- AGGREGATE and UPDATE are learnable functions, often implemented as neural networks

2.4.2 Graph Convolutional Networks (GCNs)

Graph Convolutional Networks, introduced by Kipf and Welling (2017), are a popular GNN variant that generalizes the convolution operation from regular grids (like images) to irregular graph structures. The key insight is to interpret graph convolution as neighborhood aggregation with appropriate normalization.

The layer-wise propagation rule in a GCN is:

$$\mathbf{H}^{(l+1)} = \sigma\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right) \tag{114}$$

where:

- $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ is the adjacency matrix with added self-loops (ensures each node includes itself in aggregation)
- $\tilde{\mathbf{D}}$ is the degree matrix of $\tilde{\mathbf{A}}$ (diagonal matrix with $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$)
- $\mathbf{H}^{(l)}$ is the matrix of node features/embeddings at layer l (initially $\mathbf{H}^{(0)} = \mathbf{X}$)
- $\mathbf{W}^{(l)}$ is the trainable weight matrix
- σ is a non-linear activation function (typically ReLU)

From a node-level perspective, this update can be viewed as:

$$\mathbf{h}_v^{(l+1)} = \sigma\left(\sum_{u\in\mathcal{N}(v)\cup\{v\}} \frac{1}{\sqrt{\tilde{d}_v\tilde{d}_u}}\mathbf{W}^{(l)}\mathbf{h}_u^{(l)}\right) \tag{115}$$

which is essentially a weighted average of transformed neighbor features.

GCN Step-by-Step Example

To illustrate GCN computation, consider a simple graph with 3 nodes:

- Node features: $\mathbf{X} = \begin{bmatrix}1\\2\\3\end{bmatrix}$ (each node has a 1D feature)
- Edges: (0,1) and (1,2)
- Adjacency matrix: $\mathbf{A} = \begin{bmatrix}0&1&0\\1&0&1\\0&1&0\end{bmatrix}$

First, we add self-loops:

$$\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I} = \begin{bmatrix}0&1&0\\1&0&1\\0&1&0\end{bmatrix} + \begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix} = \begin{bmatrix}1&1&0\\1&1&1\\0&1&1\end{bmatrix} \tag{116}$$

Next, compute the degree matrix:

$$\tilde{\mathbf{D}} = \text{diag}(\sum_j \tilde{A}_{ij}) = \begin{bmatrix}2&0&0\\0&3&0\\0&0&2\end{bmatrix} \tag{117}$$

Calculate the normalized adjacency:

$$\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}} = \begin{bmatrix}\frac{1}{\sqrt{2}}&0&0\\0&\frac{1}{\sqrt{3}}&0\\0&0&\frac{1}{\sqrt{2}}\end{bmatrix}\begin{bmatrix}1&1&0\\1&1&1\\0&1&1\end{bmatrix}\begin{bmatrix}\frac{1}{\sqrt{2}}&0&0\\0&\frac{1}{\sqrt{3}}&0\\0&0&\frac{1}{\sqrt{2}}\end{bmatrix} \tag{118}$$

$$= \begin{bmatrix}\frac{1}{2}&\frac{1}{\sqrt{6}}&0\\\frac{1}{\sqrt{6}}&\frac{1}{3}&\frac{1}{\sqrt{6}}\\0&\frac{1}{\sqrt{6}}&\frac{1}{2}\end{bmatrix} \tag{119}$$

With a weight matrix $\mathbf{W}^{(0)} = [2]$ (1×1 matrix for this example), the first layer update is:

$$\mathbf{H}^{(1)} = \sigma\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\mathbf{W}^{(0)}\right) \tag{120}$$

$$= \sigma\left(\begin{bmatrix} \frac{1}{2} & \frac{1}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{6}} & \frac{1}{3} & \frac{1}{\sqrt{6}} \\ 0 & \frac{1}{\sqrt{6}} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \cdot [2]\right) \tag{121}$$

$$= \sigma\left(\begin{bmatrix} \frac{1}{2} \cdot 1 + \frac{1}{\sqrt{6}} \cdot 2 + 0 \cdot 3 \\ \frac{1}{\sqrt{6}} \cdot 1 + \frac{1}{3} \cdot 2 + \frac{1}{\sqrt{6}} \cdot 3 \\ 0 \cdot 1 + \frac{1}{\sqrt{6}} \cdot 2 + \frac{1}{2} \cdot 3 \end{bmatrix} \cdot 2\right) \tag{122}$$

$$= \sigma\left(\begin{bmatrix} 1.32 \\ 2.33 \\ 3.32 \end{bmatrix}\right) \tag{123}$$

$$= \begin{bmatrix} 1.32 \\ 2.33 \\ 3.32 \end{bmatrix} \quad \text{(assuming ReLU activation, all values are positive)} \tag{124}$$

This step-by-step calculation shows how GCN updates node representations by aggregating information from neighbors. Each node's new representation is influenced by both its own features and its neighbors' features, weighted by the normalized adjacency matrix.

GCN Implementation

The following code implements a GCN layer and applies it to anomaly detection:

Listing 10 GCN Implementation for Anomaly Detection

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE

# Convert NetworkX graph to PyTorch tensors
def graph_to_torch(G):
    """
    Convert NetworkX graph to PyTorch tensors for GCN.

    Parameters:
    -----------
    G : networkx.Graph
        Input graph with node features

    Returns:
    --------
    adj_tensor : torch.Tensor
        Normalized adjacency matrix
    features_tensor : torch.Tensor
        Node features matrix
    """
    # Get adjacency matrix
    adj = nx.adjacency_matrix(G).toarray()

    # Add self-loops
    adj = adj + np.eye(adj.shape[0])

    # Normalize: D^(-1/2) A D^(-1/2)
    D_inv_sqrt = np.diag(np.power(np.sum(adj, axis=1), -0.5))
```

```
    adj_normalized = D_inv_sqrt.dot(adj).dot(D_inv_sqrt)

    # Extract node features if available
    if 'features' in G.nodes[0]:
        features = np.array([G.nodes[i]['features'] for i in range(len(G.nodes()))
            ])
    else:
        # Use one-hot encoding if no features
        features = np.eye(len(G.nodes()))

    # Convert to PyTorch tensors
    adj_tensor = torch.FloatTensor(adj_normalized)
    features_tensor = torch.FloatTensor(features)

    return adj_tensor, features_tensor

# Define GCN Layer
class GCNLayer(nn.Module):
    """
    Graph Convolutional Network layer.
    """
    def __init__(self, in_features, out_features):
        super(GCNLayer, self).__init__()
        self.linear = nn.Linear(in_features, out_features)

    def forward(self, x, adj):
        # Graph convolution: H^(l+1) = σ(D^-1/2 A D^-1/2 H^(l) W^(l))
        support = torch.mm(x, self.linear.weight.t()) + self.linear.bias
        output = torch.mm(adj, support)
        return output

# Define GCN Autoencoder model
class GCNAutoencoder(nn.Module):
    """
    Graph Convolutional Network Autoencoder for anomaly detection.
    """
    def __init__(self, input_dim, hidden_dims, latent_dim):
        super(GCNAutoencoder, self).__init__()

        # Encoder layers
        self.encoder_layers = nn.ModuleList()
        prev_dim = input_dim
        for dim in hidden_dims:
            self.encoder_layers.append(GCNLayer(prev_dim, dim))
            prev_dim = dim
        self.encoder_layers.append(GCNLayer(prev_dim, latent_dim))

        # Decoder for structure reconstruction (inner product)
        self.decoder_struct = nn.Linear(latent_dim, input_dim)

    def forward(self, x, adj):
        # Encoder
        z = x
        for layer in self.encoder_layers:
            z = F.relu(layer(z, adj))

        # Structure reconstruction
        adj_hat = torch.sigmoid(torch.mm(z, z.t()))

        # Feature reconstruction
        x_hat = self.decoder_struct(z)

        return z, adj_hat, x_hat

    def encode(self, x, adj):
        """Get latent representations"""
        z = x
        for layer in self.encoder_layers:
```

```
            z = F.relu(layer(z, adj))
        return z

def gcn_anomaly_detection(G, hidden_dims=[32, 16], latent_dim=8,
                          num_epochs=200, learning_rate=0.01, alpha=0.5,
                          contamination=0.1):
    """
    Detect anomalies using GCN Autoencoder.

    Parameters:
    -----------
    G : networkx.Graph
        Input graph with node features
    hidden_dims : list
        Dimensions of hidden layers
    latent_dim : int
        Dimension of latent representation
    num_epochs : int
        Number of training epochs
    learning_rate : float
        Learning rate for optimization
    alpha : float
        Weight for structure vs. feature reconstruction loss
    contamination : float
        Expected proportion of anomalies

    Returns:
    --------
    anomalies : set
        Set of anomalous node indices
    scores : dict
        Dictionary of anomaly scores for each node
    embeddings : numpy.ndarray
        Node embeddings from GCN
    """
    # Convert graph to tensors
    adj_tensor, features_tensor = graph_to_torch(G)

    # Initialize model
    n_features = features_tensor.shape[1]
    model = GCNAutoencoder(input_dim=n_features, hidden_dims=hidden_dims,
         latent_dim=latent_dim)
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    # Train model
    model.train()
    for epoch in range(num_epochs):
        optimizer.zero_grad()

        # Forward pass
        z, adj_hat, x_hat = model(features_tensor, adj_tensor)

        # Compute loss (structure and feature reconstruction)
        struct_loss = F.binary_cross_entropy(adj_hat, adj_tensor)
        feat_loss = F.mse_loss(x_hat, features_tensor)
        loss = alpha * struct_loss + (1 - alpha) * feat_loss

        # Backward pass and optimize
        loss.backward()
        optimizer.step()

        # Print progress
        if (epoch + 1) % 50 == 0:
            print(f'Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.4f}')

    # Calculate anomaly scores
    model.eval()
    with torch.no_grad():
```

```
        z, adj_hat, x_hat = model(features_tensor, adj_tensor)

        # Structure reconstruction error
        struct_error = torch.sum((adj_tensor - adj_hat) ** 2, dim=1)

        # Feature reconstruction error
        feat_error = torch.sum((features_tensor - x_hat) ** 2, dim=1)

        # Combined anomaly score
        anomaly_scores = alpha * struct_error + (1 - alpha) * feat_error

    # Convert to numpy
    scores_np = anomaly_scores.numpy()
    embeddings_np = z.numpy()

    # Identify anomalies (top contamination% by score)
    threshold = np.percentile(scores_np, 100 * (1 - contamination))
    is_anomaly = scores_np > threshold
    anomalies = set(np.where(is_anomaly)[0])

    # Create scores dictionary
    scores = {node: scores_np[node] for node in G.nodes()}

    return anomalies, scores, embeddings_np

def visualize_gcn_results(G, anomalies, embeddings, scores):
    """
    Visualize GCN anomaly detection results.

    Parameters:
    -----------
    G : networkx.Graph
        Input graph
    anomalies : set
        Set of anomalous nodes
    embeddings : numpy.ndarray
        Node embeddings from GCN
    scores : dict
        Dictionary of anomaly scores
    """
    # Convert scores to list in node order
    score_values = [scores[node] for node in sorted(G.nodes())]

    # Reduce embeddings to 2D with t-SNE
    tsne = TSNE(n_components=2, perplexity=min(30, len(G.nodes())-1), random_state
        =42)
    embeddings_2d = tsne.fit_transform(embeddings)

    # Create node colors based on anomaly status
    node_colors = ['red' if node in anomalies else 'skyblue' for node in G.nodes()]

    # Create figure
    plt.figure(figsize=(18, 6))

    # Original graph
    plt.subplot(131)
    pos = nx.spring_layout(G, seed=42)
    nx.draw(G, pos, node_color=node_colors, with_labels=True)
    plt.title('Original Graph with Detected Anomalies')

    # Embedding visualization
    plt.subplot(132)
    scatter = plt.scatter(
        embeddings_2d[:, 0],
        embeddings_2d[:, 1],
        c=score_values,
        cmap='YlOrRd',
        s=100,
```

```
        alpha=0.8
    )
    plt.colorbar(scatter, label='Anomaly Score')

    # Add node labels
    for i, (x, y) in enumerate(embeddings_2d):
        plt.text(x, y, str(i), fontsize=9)

    plt.title('GCN Embeddings (t-SNE)')
    plt.grid(True, alpha=0.3)

    # Anomaly score distribution
    plt.subplot(133)
    plt.hist(score_values, bins=20, alpha=0.7)
    plt.axvline(x=min([scores[node] for node in anomalies]), color='r', linestyle='--',
                label='Anomaly Threshold')
    plt.title('Anomaly Score Distribution')
    plt.xlabel('Anomaly Score')
    plt.ylabel('Frequency')
    plt.legend()

    plt.tight_layout()
    plt.show()

# Example usage
gcn_anomalies, gcn_scores, gcn_embeddings = gcn_anomaly_detection(
    sample_graph, hidden_dims=[8], latent_dim=4, num_epochs=300)
print(f"GCN detected anomalies: {gcn_anomalies}")
visualize_gcn_results(sample_graph, gcn_anomalies, gcn_embeddings, gcn_scores)
```

This implementation demonstrates several important aspects of GCN-based anomaly detection:

- Creating properly normalized adjacency matrices with self-loops
- Implementing the GCN layer that performs the core graph convolution operation
- Building a GCN autoencoder model for both structure and feature reconstruction
- Computing anomaly scores based on reconstruction error
- Visualizing the results, including the embeddings learned by the GCN

2.4.3 GCN Autoencoder for Anomaly Detection

Graph autoencoders combine GNN-based encoders with decoders that reconstruct graph structure, node features, or both. For anomaly detection, we train the autoencoder on a graph presumed to contain primarily normal nodes, then identify nodes with high reconstruction error as potential anomalies.

Architecture

A typical GCN-based autoencoder has:

- **Encoder:** GCN layers that map the input graph to node embeddings $Z = f_{enc}(A, X)$
- **Structure Decoder:** Reconstructs adjacency matrix via $\hat{A} = \sigma(ZZ^T)$
- **Feature Decoder:** Reconstructs node features via $\hat{X} = g_{dec}(Z)$, often using a simple MLP

Loss Functions

The training objective combines structure and feature reconstruction:

$$\mathcal{L}_{struct} = \|A - \hat{A}\|_F^2 \text{ or } -\sum_{i,j}[A_{ij}\log(\hat{A}_{ij}) + (1 - A_{ij})\log(1 - \hat{A}_{ij})] \tag{125}$$

$$\mathcal{L}_{feat} = \|X - \hat{X}\|_F^2 \tag{126}$$

$$\mathcal{L} = \alpha\mathcal{L}_{struct} + (1 - \alpha)\mathcal{L}_{feat} \tag{127}$$

where $\alpha \in [0, 1]$ controls the relative importance of structure versus feature reconstruction.

Anomaly Scoring

After training, we compute node-level anomaly scores based on reconstruction error:

$$s_{struct}(v) = \sum_u (A_{vu} - \hat{A}_{vu})^2 \tag{128}$$

$$s_{feat}(v) = \|X_v - \hat{X}_v\|^2 \tag{129}$$

$$s(v) = \alpha \cdot s_{struct}(v) + (1 - \alpha) \cdot s_{feat}(v) \tag{130}$$

Nodes with the highest scores are flagged as potential anomalies.

Advanced Variations

Several extensions have been proposed to improve GCN-based anomaly detection:

- **Variational Graph Autoencoders (VGAE):** Add regularization via a variational inference framework
- **Adversarial Training:** Incorporate adversarial components to improve robustness
- **Attention Mechanisms:** Use graph attention networks (GAT) instead of GCN for more flexible aggregation
- **Edge-Enhanced Reconstruction:** Consider edge features in addition to node features

3 Comparative Evaluation and Applications

Having explored various methods for both time series and graph-based anomaly detection, we now compare their relative strengths and limitations and examine their application across different domains.

3.1 Comparative Analysis

The following table summarizes the key characteristics and trade-offs of the anomaly detection methods discussed in this paper:

Table 1 Comparative Analysis of Anomaly Detection Methods

Method Class	Computational Complexity	Feature Handling	Handling Long Dependencies	Interpretability
Statistical (Window-based)	Low	Limited	Poor	High
ARIMA	Medium	Limited	Medium	Medium
Feedforward Networks	Medium	Good	Poor	Low
RNNs	High	Good	Limited	Low
LSTMs	High	Excellent	Excellent	Low
LSTM Autoencoders	High	Excellent	Excellent	Medium
Statistical Graph Methods	Low-Medium	Limited	N/A	High
Graph Embeddings	Medium	Medium	N/A	Low
GCNs	High	Excellent	N/A	Low
GCN Autoencoders	High	Excellent	N/A	Medium

Key Trade-offs

- **Computational Resources vs. Model Power:** More complex models like LSTMs and GCNs require substantial computational resources but can capture more complex patterns
- **Interpretability vs. Performance:** Statistical methods offer clearer interpretability but often underperform deep learning approaches on complex data
- **Training Data Requirements vs. Generalization:** Deep learning methods require more training data but generalize better to unseen patterns
- **Specificity vs. Versatility:** Domain-specific statistical methods may excel on particular data types but lack the versatility of deep learning approaches

3.2 Hybrid Approaches

Hybrid approaches that combine multiple methods can leverage their complementary strengths:

- **Statistical + Deep Learning:** Use statistical methods as preprocessing or feature engineering for deep learning models
- **Time Series + Graph Methods:** Combine both representations for data with both temporal and relational aspects (e.g., evolving networks)
- **Ensemble Models:** Combine predictions from multiple models for more robust anomaly detection

Case Study: Bitcoin Transaction Network Analysis

The Elliptic dataset contains a Bitcoin transaction graph with over 200,000 transactions (nodes) and 234,000 payment flows (edges). Each node has 166 features related

to the transaction. A subset of transactions is labeled as licit (legal) or illicit (e.g., money laundering, scams).

Using GCN-based anomaly detection:

- The graph structure revealed connected components of illicit transactions
- Temporal evolution of transaction patterns helped identify emerging fraud techniques
- The model achieved 95% AUC in distinguishing illicit from licit transactions
- Key anomalous patterns included cyclic transaction structures and rapid fan-out transactions