High Integrity Systems Project Time Series Analysis

Important Findings

Mehjabeen Jahangeer Khan
Aniket Nighot
Chitra Khatri
Hellyben Shah
Zain Hanif
Aaliya Javed
Hardikumar Khunt

February 13, 2025

Why Multi-Step Forecasting?

In most real-world situations, we need to predict multiple future time points rather than just the next one. Whether it is tracking household energy consumption or forecasting product sales, businesses and individuals rely on these predictions to make informed decisions.

Despite its importance, multi-step forecasting has not received as much attention as it should. One reason is that traditional statistical models like ARIMA and exponential smoothing already include built-in methods for handling multi-step predictions, making them easy to use. Another reason is that forecasting multiple steps ahead is more challenging—uncertainty increases the further we predict into the future, making accurate results harder to achieve.

How to Choose a Multi-Step Forecasting Strategy

Selecting the right multi-step forecasting approach depends on several factors, including the nature of the data, forecasting objectives, and the required level of accuracy. Below are key strategies to consider:

Direct Method

- Builds a separate model for each future time step.
- Useful when each forecasted step has unique patterns.
- Requires more data and training but avoids error accumulation.

Recursive Method

- Uses a single model to predict the next step, then feeds that prediction back as input for future steps.
- Simple to implement but prone to error propagation as predictions influence future steps.

Hybrid (Direct-Recursive) Method

- Combines elements of both direct and recursive approaches.
- Can balance accuracy and computational efficiency.

Multi-Output Model

- Trains a single model to predict multiple future steps at once.
- Works well for deep learning approaches and avoids sequential error buildup.

Factors to Consider When Choosing a Strategy

- **Data Availability:** More complex methods require more data.
- **Computational Power:** Deep learning-based multi-output models demand more resources.
- **Accuracy Needs:** Direct methods tend to be more accurate but require more effort.

- **Forecast Horizon:** Short-term forecasts might work well with recursive methods, while long-term ones need more robust techniques.

Key Components of Multi-Step Forecasting

Multi-step forecasting involves predicting multiple future time steps instead of just the next one. To ensure accurate and reliable forecasts, several key components must be considered:

1. Forecasting Strategy

Choosing the right approach is crucial. The main strategies include:

- **Direct Method:** Builds a separate model for each future time step.
- **Recursive Method:** Uses previous predictions as inputs for future steps.
- **Hybrid Method:** Combines both direct and recursive approaches.
- **Multi-Output Model:** Predicts multiple future steps at once.

2. Model Selection

Selecting an appropriate model depends on data patterns and computational resources. Common models include:

- **Statistical Models:** ARIMA, Exponential Smoothing, Vector Autoregression (VAR).
- **Machine Learning Models:** Random Forest, Gradient Boosting, Support Vector Machines.
- **Deep Learning Models:** Long Short-Term Memory (LSTM), Transformers, Convolutional Neural Networks (CNNs).

3. Feature Engineering

To improve forecasting accuracy, feature selection and transformation play a vital role:

- **Lag Features:** Using past values as predictors.
- **Rolling Statistics:** Moving averages, standard deviations.
- **Seasonal and Trend Features:** Capturing periodic patterns.
- **Exogenous Variables:** External factors such as weather, economic indicators.

These components are essential for building effective multi-step forecasting models, ensuring they are accurate, robust, and adaptable for practical applications.

Limitations of Multi-Step Forecasting

Despite its usefulness, multi-step forecasting has several challenges:

- **Error Accumulation:** Forecast errors compound as predictions are used for future steps.
- **Increased Uncertainty:** The further into the future we predict, the less reliable the results become.
- **Model Complexity:** More sophisticated models are often required, increasing computational costs.
- **Data Dependency:** Accuracy depends heavily on high-quality, sufficient historical data.
- **Computational Cost:** Long-term forecasting requires significant processing power and time.

These challenges make multi-step forecasting more difficult compared to single-step predictions, requiring careful model selection and tuning.

Core Principles of the Kalman Filter

The Kalman Filter operates on several key principles, which give it the ability to estimate the hidden state of a dynamic system:

- **State Estimation in Dynamic Systems**: At each time step, the Kalman Filter estimates the system's current state and the uncertainty in that estimate. It then updates these estimates when new measurements become available, balancing the prediction with the new information.
- **Optimality Under Gaussian Assumptions**: The Kalman Filter is mathematically optimal when:
  - The system dynamics are **linear**,
  - The process and measurement noise are **Gaussian** with zero mean and known covariances. Under these assumptions, the Kalman Filter minimizes the mean squared error of the estimated state.
- **Recursive Process**: The Kalman Filter is recursive, meaning that it only requires the previous state estimate and measurement to generate the current estimate. This recursive property makes it highly efficient for real-time applications, as it avoids the need to store extensive historical data.
- **Minimizing the Covariance (Error Uncertainty)**: The Kalman Filter is designed to minimize the error covariance, which quantifies the uncertainty in the state estimate. This is done by adjusting the weight (Kalman Gain) between the prediction and the measurement at each time step.

Kalman Filter Algorithm

The Kalman filter operates in a predict-update (correction) cycle.

Prediction Step

State Prediction:

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_k$$

This predicts the next state using the previous estimate.

Error Covariance Prediction:

$$P_k^- = AP_{k-1}A^T + Q$$

This propagates the uncertainty in the estimate.

Update (Correction) Step

Kalman Gain Computation:

$$K_k = P_k^- H^T \left(HP_k^- H^T + R\right)^{-1}$$

The Kalman gain determines how much the new measurement should influence the state estimate.

State Update (Correction):

$$\hat{x}_k = \hat{x}_k^- + K_k \left(z_k - H\hat{x}_k^-\right)$$

The estimate is corrected using the measurement residual $(z_k - H\hat{x}_k^-)$.

Error Covariance Update:

$$P_k = (I - K_k H)P_k^-$$

This updates the estimate's uncertainty.

Different Variants of the Kalman Filter

- **Extended Kalman Filter (EKF):** Designed for non-linear systems by linearizing the process and measurement functions around the current estimate.
- **Unscented Kalman Filter (UKF):** Utilizes a set of carefully chosen sample points to more accurately capture the mean and covariance, better suited for highly non-linear systems than EKF.
- **Ensemble Kalman Filter (EnKF):** Useful for large-scale systems, it propagates an ensemble of estimates instead of covariance, commonly applied in geophysical models.
- **Information Filter:** An alternative formulation using the information matrix (inverse of covariance), which can sometimes simplify computation, especially in multi-sensor networks.
- **Square-Root Kalman Filter:** Improves numerical stability by calculating square roots of covariance matrices instead of the covariance matrix directly.

Data-Centric Workflow for Model Development

Below are the basic steps typically followed in deep learning models like CNN, LSTM, and CNN-LSTM for classification or prediction tasks:

1. **Data Collection:** Collect and gather the relevant data for the task.
2. **Data Preprocessing:** Preprocess the data to make it suitable for input into the model.
    - Remove noise and irrelevant data.
    - Handle missing values (e.g., imputation or deletion).
    - Rescale or normalize features.
    - Data augmentation (especially in CNNs for image data).
3. **Data Splitting:** Split the data into training, validation, and test sets.
4. **Model Initialization:** Initialize the model architecture.
    - For CNN: Initialize convolutional layers, pooling layers, fully connected layers, etc.
    - For LSTM: Initialize LSTM layers, input-output layers, and regularization (e.g., dropout).
    - For CNN-LSTM: Combine CNN for feature extraction followed by LSTM layers for sequence modeling.
5. **Model Compilation:** Choose the appropriate loss function, optimizer, and evaluation metrics.
    - Loss function: E.g., categorical cross-entropy for classification.
    - Optimizer: E.g., Adam, SGD, RMSProp.
    - Metrics: Accuracy, precision, recall, etc.
6. **Model Training:** Train the model using the training dataset.
    - Feed data through the model.
    - Perform forward propagation and compute the loss.
    - Perform backpropagation to adjust weights using gradient descent or variants.
    - Monitor the model's performance using validation data.
7. **Model Evaluation:** Evaluate the trained model on the test set.
    - Calculate performance metrics (accuracy, F1-score, etc.).
    - Adjust hyperparameters or architecture if needed.
8. **Prediction/Classification:** Once the model is trained, make predictions on new, unseen data.
    - Pass the data through the trained model.
    - Get the output and interpret the results (e.g., class labels or continuous values).
9. **Model Deployment:** Deploy the model into production for real-time predictions or batch processing.

Convolutional Neural Networks (CNN)

Core Idea

Convolutional Neural Networks (CNNs) are designed to automatically and adaptively learn spatial hierarchies of features from input data. They are particularly powerful for tasks involving grid-like data such as images, time-series data like ECG signals, and more. CNNs operate through a series of layers that capture increasingly complex features of the input data.

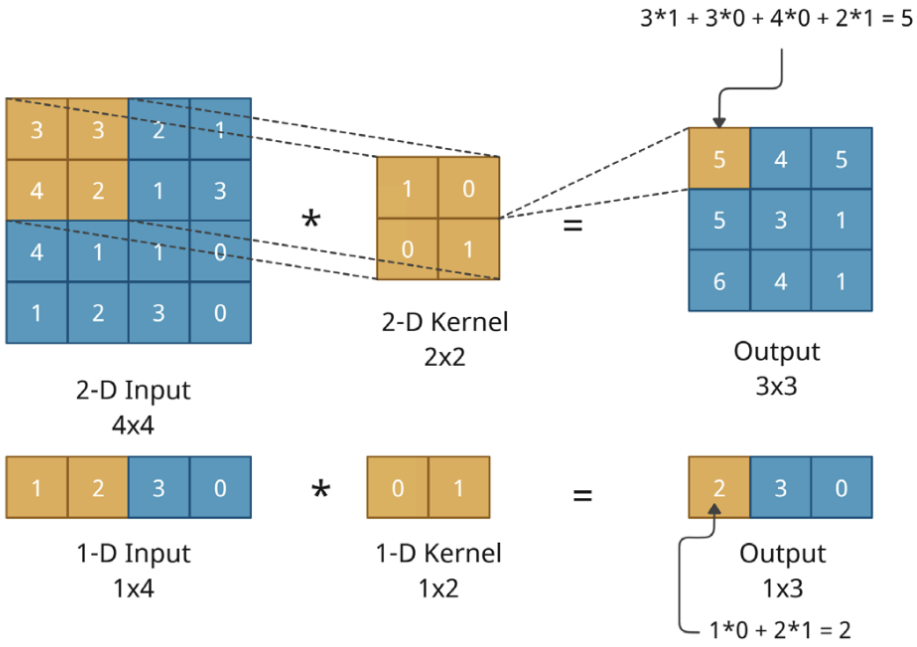

Figure 1: A convolution operation on 2D and 1D inputs [1].

Inputs to a CNN

A CNN takes in the following inputs:

1. **Current Input (x_t):** This is the raw data, such as an ECG signal or an image. The input data can have multiple dimensions (e.g., 2D for images or 1D for ECG signals).

2. **Filters/Kernels (w):** These are small learnable weights that slide over the input data to extract features such as edges or patterns. The filters are shared across the entire input, which reduces the number of parameters.

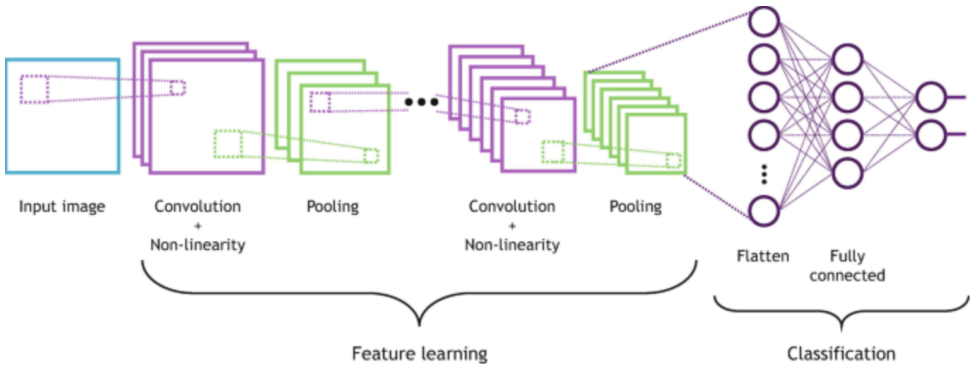

Figure 2: Inputs to CNN.

What Happens to the Input

Once the input data is fed into the CNN, the following steps occur at each layer:

1. **Convolution Operation:** The input data is convolved with the filters (kernels) to produce feature maps. This operation detects basic features in the data, like edges, corners, or other patterns. The formula for the convolution operation is:

$$y_i = f\left(\sum_{j=0}^{k-1} w_j x_{i+j} + b\right)$$

   where:

   - x represents the input sequence (e.g., ECG signal),
   - w denotes the kernel weights (filters),
   - b is the bias term,
   - f is the activation function (such as ReLU),
   - k is the kernel size.

2. **Activation Function:** After convolution, the output is passed through an activation function like ReLU to introduce non-linearity. This enables the network to learn more complex patterns.

3. **Pooling Layer:** Pooling operations like max pooling or average pooling are applied to reduce the spatial dimensions of the feature maps, thereby reducing computational complexity while retaining important features.

4. **Fully Connected Layer:** The pooled feature maps are flattened and passed through fully connected layers, where the final output is generated (classification or regression).

Components of a CNN

A CNN typically consists of the following layers:

- **Convolutional Layers:** Extract spatial features from the input (e.g., ECG signals) using filters (kernels). These layers detect patterns like edges, textures, and other important features.

- **Activation Functions:** Non-linear functions such as ReLU (Rectified Linear Unit) are used to introduce non-linearity into the model, allowing it to learn complex patterns.
- **Pooling Layers:** Reduce the spatial dimensions (downsampling), decreasing the computational cost and the risk of overfitting while maintaining important information.
- **Fully Connected Layers:** These layers make final predictions based on the learned features, typically leading to classification or regression outcomes.

Advantages of CNNs

CNNs offer several advantages, particularly for tasks involving spatial or temporal data:

- **Efficient Feature Extraction:** CNNs automatically learn spatial patterns from raw input data without manual feature engineering.
- **Reduced Number of Parameters:** By using shared weights in convolutional filters, CNNs significantly reduce the number of parameters compared to fully connected networks, leading to lower computational complexity.
- **Robustness:** CNNs are highly robust to noise and distortions, making them suitable for real-world data with imperfections.
- **Scalability:** CNNs can scale well to large datasets, enabling effective learning from massive amounts of data.

Limitations of CNNs

Despite their strengths, CNNs come with certain limitations:

- **Large Datasets Required:** CNNs typically require large datasets for training to avoid overfitting and achieve good generalization.
- **Computational Intensity:** CNNs are computationally expensive, requiring significant resources, including GPUs, for efficient training and inference.
- **Complex Hyperparameter Tuning:** The process of tuning CNN hyperparameters (such as kernel size, number of layers, and learning rate) can be time-consuming and requires extensive experimentation.
- **Interpretability:** CNNs lack the interpretability of traditional machine learning models, making it harder to understand why a model makes certain predictions.

Applications of CNNs

CNNs are widely used in a variety of domains, particularly in the analysis of sequential or image data:

- **Arrhythmia Detection:** CNNs can be used to detect and classify arrhythmias in ECG signals by learning features that correspond to abnormal heart patterns.

- **Heart Condition Prediction:** CNNs can predict future heart conditions based on historical ECG data, enabling early intervention.
- **Wearable Health Monitoring:** CNNs are applied in real-time ECG monitoring systems integrated into wearable devices, providing continuous health tracking.
- **Noise and Artifact Removal:** CNNs are effective in removing noise and artifacts from ECG signals, improving the quality of the input data for further analysis.

Long Short-Term Memory (LSTM) Networks

Core Idea

Long Short-Term Memory (LSTM) networks are an advanced version of Recurrent Neural Networks (RNNs) introduced by Hochreiter and Schmidhuber in 1997. LSTMs solve key issues of vanishing and exploding gradients in vanilla RNNs. Inspired by logic gates in computers, LSTMs incorporate a special component called a **memory cell**, which acts as long-term memory, in addition to the hidden state used in classical RNNs.

Inputs to an LSTM

An LSTM network processes the following inputs:

1. **Current Input (X_t):** This is the raw data (e.g., ECG signal, time-series data, or text) at the current time step.
2. **Previous Hidden State (H_{t-1}):** This is the output of the previous LSTM cell at the previous time step, which carries information from past time steps.
3. **Previous Cell State (C_{t-1}):** This is the long-term memory from the previous time step, which stores important past information.

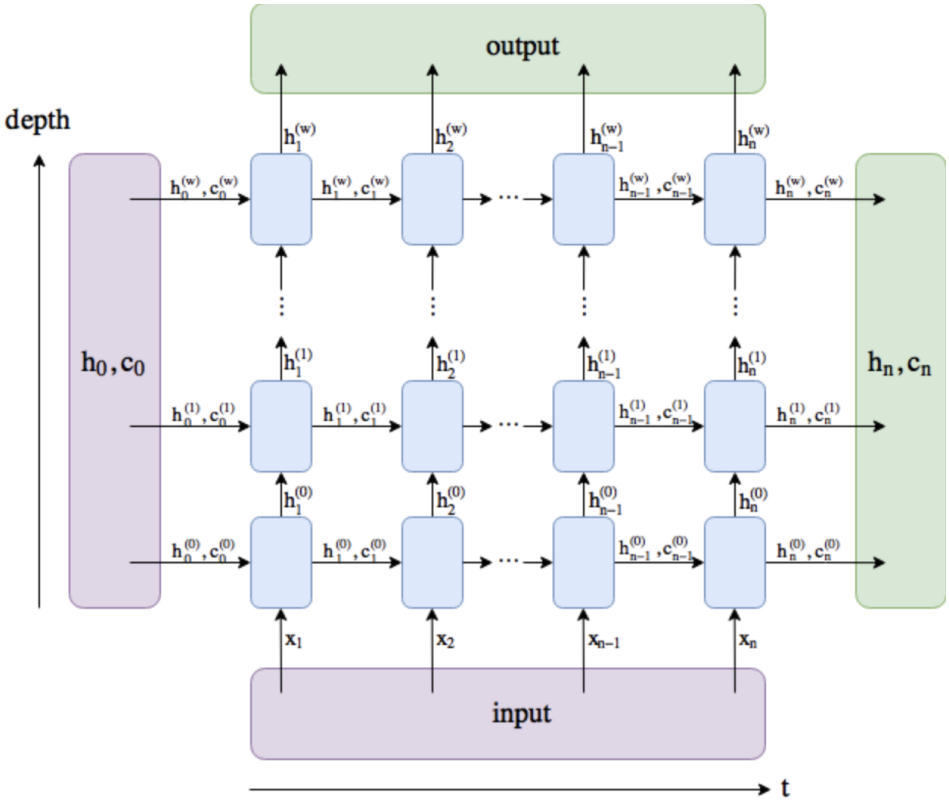

Figure 3: Inputs to LSTM.

What Happens to the Input

At each time step, the LSTM processes the input data through its various gates and updates the hidden and cell states:

1. **Input Gate:** This gate decides how much of the new information from the current input and previous hidden state should be stored in the memory cell. The input gate's formula is:

$$I_t = \sigma(W_{xi} \cdot X_t + W_{hi} \cdot H_{t-1} + b_i)$$

2. **Forget Gate:** This gate determines how much of the previous memory (C_{t-1}) should be forgotten. The forget gate's formula is:

$$F_t = \sigma(W_{xf} \cdot X_t + W_{hf} \cdot H_{t-1} + b_f)$$

3. **Output Gate:** This gate decides what part of the memory should be output at the current time step. The output gate's formula is:

$$O_t = \sigma(W_{xo} \cdot X_t + W_{ho} \cdot H_{t-1} + b_o)$$

4. **Cell State Update:** This step updates the memory cell. The candidate cell state ($\tilde{C}_t$) is first calculated, and then it is combined with the forget gate to form the updated cell state (C_t):

$$\tilde{C}_t = \tanh(W_{xc} \cdot X_t + W_{hc} \cdot H_{t-1} + b_c)$$

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t$$

5. **Hidden State Update:** The updated memory cell is then passed through the output gate, and the new hidden state (H_t) is calculated:

$$H_t = O_t \odot \tanh(C_t)$$

In the above equations: - σ is the sigmoid activation function. - $\odot$ denotes element-wise multiplication. - $W_{xi}, W_{hi}, W_{xf}, W_{hf}, W_{xo}, W_{ho}, W_{xc}, W_{hc}$ represent weight matrices for the respective gates and the cell state update. - b_i, b_f, b_o, b_c are bias terms for the respective gates.

Key Features of LSTMs

LSTMs are specifically designed to handle long-term dependencies in sequential data. The main features of LSTMs include:

- **Memory Cell:** Acts like long-term memory, storing crucial information over long periods.
- **Gates:** LSTMs utilize three main gates (Input, Forget, and Output gates) to control how information flows through the network and how the memory is updated.
  - **Input Gate:** Decides how much new information from the input should be stored.
  - **Forget Gate:** Decides how much information from the previous time step should be discarded.
  - **Output Gate:** Determines what information from the memory cell should be output at each time step.

Why LSTMs are Important

LSTMs are crucial for tasks that involve sequential data, such as time-series data, text, and speech, due to their ability to capture long-term dependencies. Their importance lies in the following areas:

- **Language Modeling and Translation:** LSTMs can predict the next word in a sentence or translate text from one language to another by remembering long-range context.
- **Speech Recognition:** LSTMs excel at handling speech signals where context over time is essential for accurate recognition.
- **Stock Price Prediction:** LSTMs can model time-series data like stock prices, making them valuable for financial forecasting.
- **General Sequence Prediction:** LSTMs are broadly applicable to any domain that involves sequential data, such as ECG signals, video frames, and more.

Architecture of LSTMs

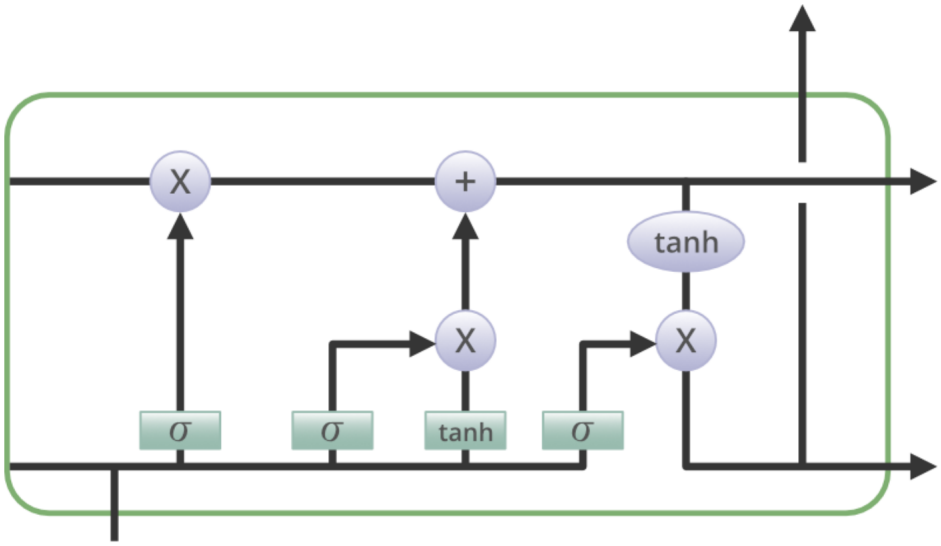

Figure 4: Diagram of a Long Short-Term Memory (LSTM) cell architecture [2].

Advantages of LSTMs

LSTMs have several advantages that make them suitable for a wide range of applications:

- **Long-Term Dependencies:** LSTMs are designed to capture long-term dependencies in time-series and sequential data.
- **Vanishing Gradient Problem:** LSTMs address the vanishing gradient problem, which occurs in traditional RNNs, allowing them to learn from long sequences.
- **Versatile Applications:** LSTMs are effective for a variety of tasks, including speech, language, and time-series data analysis.
- **Variable-Length Sequences:** LSTMs can handle variable-length sequences, which makes them adaptable to different types of input data.

Limitations of LSTMs

Despite their advantages, LSTMs have certain limitations:

- **Computational Complexity:** LSTMs are computationally expensive due to their complex gate mechanisms.
- **Training Time:** Training LSTM models requires significant time and resources compared to simpler models.
- **Memory Requirements:** LSTMs require substantial memory for storing parameters, which can be a limitation for large-scale applications.

- **Struggles with Very Long Sequences:** While LSTMs address some issues of traditional RNNs, they may still face challenges when dealing with extremely long sequences.

Applications of LSTMs in ECG Analysis

LSTMs have demonstrated their usefulness in the analysis of ECG signals for various medical and healthcare applications:

- **ECG Signal Forecasting:** LSTMs can be used to predict future ECG signals, providing insights into the patient's condition over time.
- **Arrhythmia Classification:** LSTMs are effective in detecting and classifying arrhythmias from ECG signals, helping in the diagnosis of heart conditions.
- **Real-Time Monitoring:** LSTMs are applied in wearable devices for real-time monitoring of ECG signals, offering predictive diagnostics.
- **Feature Extraction:** LSTMs are capable of extracting important features from ECG signals, improving the accuracy of cardiovascular analysis.

Hybrid Model CNN+LSTM

The CNN-LSTM hybrid model combines the strengths of Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks to improve ECG signal forecasting. CNNs are effective in extracting spatial features, while LSTMs are powerful in capturing temporal dependencies. By integrating both architectures, the CNN-LSTM model enhances the performance of ECG signal prediction.

Inputs to the CNN-LSTM Hybrid Model

The input to the hybrid model consists of multi-lead ECG signals, where each lead captures the heart's electrical activity over time. These raw signals represent time-series data and are passed through the model for further processing.

ECG Input Data: The multi-lead ECG signals are provided as input to the model. Each lead records the electrical activity of the heart from a different angle, resulting in multiple time-series data streams.

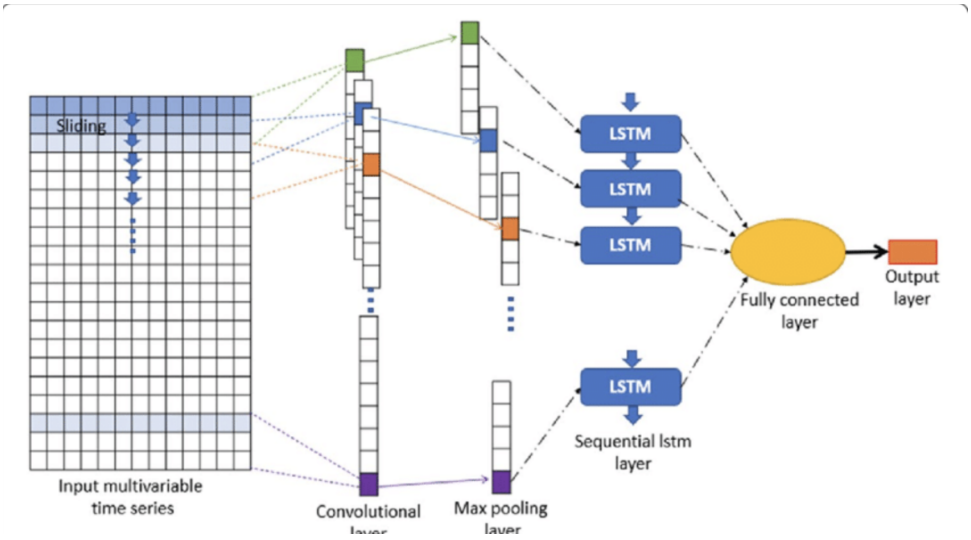

Figure 5: Inputs to CNN-LSTM Hybrid Model.

What Happens to the Input in the CNN Layers

The input ECG signal undergoes several transformations in the CNN part of the hybrid model, which processes spatial features. The CNN layers perform the following operations:

Convolutional Layers : The multi-lead ECG signals are passed through convolutional layers, which apply filters (kernels) to extract spatial features. These filters detect patterns such as peaks, QRS complexes, or ST segments from the raw ECG signals.

The 1D convolution operation is represented mathematically as:

$$y_i = f\left(\sum_{j=0}^{k-1} w_j x_{i+j} + b\right)$$

where:

- x is the ECG input signal,
- w are the kernel weights,
- b is the bias term,
- f is an activation function applied to introduce non-linearity (e.g., ReLU).

Pooling Layers: After the convolutional layers, pooling layers (typically max pooling) are applied. Pooling reduces the spatial dimensions of the output feature maps, which helps in reducing the computational cost and the number of parameters. It also makes the model more robust by retaining important spatial features while discarding less significant ones.

What Happens After the CNN Layers

Once the input has passed through the convolutional and pooling layers, the output is a set of features that represent the spatial patterns in the ECG signal. These features are then prepared for the next step in the model:

Flattening: The output of the CNN layers, which is typically in the form of a multi-dimensional tensor, is flattened into a one-dimensional vector so that it can be fed into the LSTM layers.

Fully Connected Layers: The flattened feature vector is passed to fully connected layers for final decision-making. These layers combine the extracted features to generate predictions or classifications, depending on the task.

Key Components of the CNN-LSTM Hybrid Model

The CNN-LSTM hybrid model consists of the following key components:

- **Convolutional Layers**: Extract spatial features from multi-lead ECG signals by applying convolution operations.
- **Pooling Layers**: Reduce the dimensionality of the extracted features, thereby making the model more efficient and less prone to overfitting.
- **Fully Connected Layers**: These layers use the processed features to make final predictions about the ECG signal.

Architecture of CNN-LSTM Model

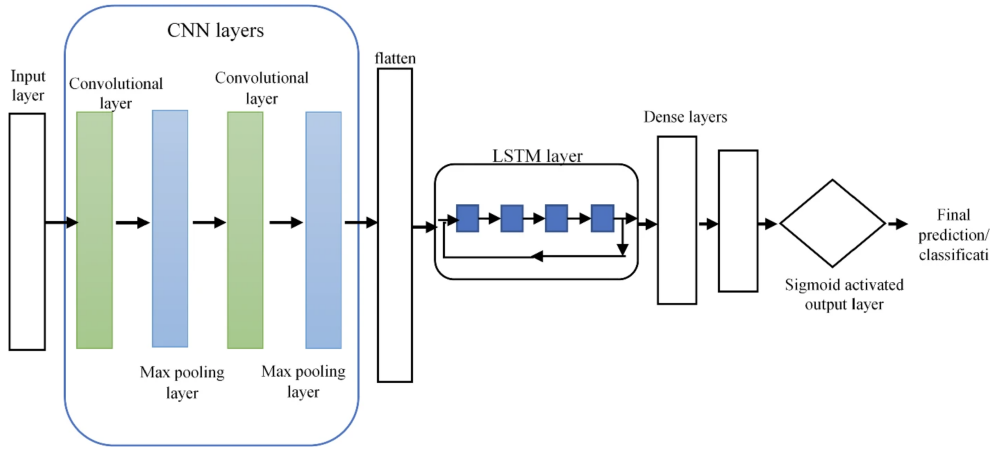

Figure 6: Structure of Proposed Hybrid CNN-LSTM model [3].

Advantages of CNN-LSTM Hybrid Model

The combination of CNNs and LSTMs offers several advantages:

- **Combines Spatial and Temporal Feature Extraction**: The CNN part extracts important spatial features, while the LSTM part (not detailed here) handles temporal dependencies in the data.
- **Reduces Computational Cost**: The use of CNNs to extract features reduces the complexity and computational cost compared to using only LSTMs.
- **Effective for Multi-Lead ECG Signals**: The model handles complex, multi-lead ECG data well, capturing rich spatial information across different leads.

- **Suitable for Real-Time Analysis**: The hybrid architecture enables real-time ECG analysis and forecasting by extracting meaningful features efficiently.

Disadvantages of CNN-LSTM Hybrid Model

While the CNN-LSTM model offers great potential, there are challenges to consider:

- **Increased Model Complexity**: The dual architecture (CNN + LSTM) adds complexity, which can make the model harder to train and tune.
- **Hyperparameter Tuning**: The interaction between CNN and LSTM components requires careful tuning of hyperparameters for optimal performance.
- **Higher Training Time**: The hybrid model requires more training time than individual models due to its increased complexity.
- **Large ECG Datasets Required**: The model needs a substantial amount of labeled ECG data to perform optimally and avoid overfitting.

Applications of CNN-LSTM in ECG Analysis

The CNN-LSTM hybrid model is highly effective for various ECG analysis tasks:

- **Advanced ECG Signal Forecasting and Anomaly Detection**: The model can predict future ECG signals and identify abnormal patterns indicative of heart disease or other conditions.
- **Detection of Arrhythmias and Cardiovascular Abnormalities**: The model classifies different arrhythmias and abnormalities in the ECG, helping in diagnosis.
- **Wearable Device Monitoring**: The model can be integrated into wearable health devices for continuous, real-time monitoring of the heart's electrical activity.
- **Automated Feature Extraction for ECG Classification**: The model automatically extracts important features, improving the accuracy of ECG classification tasks for various cardiac conditions.

Conclusion

The CNN-LSTM hybrid model effectively captures both spatial and temporal dependencies in ECG data, making it a powerful tool for ECG signal forecasting. By combining the strengths of CNNs for feature extraction and LSTMs for sequence modeling, the model enhances prediction accuracy in healthcare applications. Despite its computational intensity, the CNN-LSTM hybrid model offers promising potential for real-time ECG analysis and healthcare monitoring solutions.

Recurrent Neural Networks (RNN)

Core Idea

RNNs are designed to process sequential data by maintaining a "memory" of past inputs. Unlike traditional feedforward neural networks, RNNs have loops that allow information to persist across time steps.

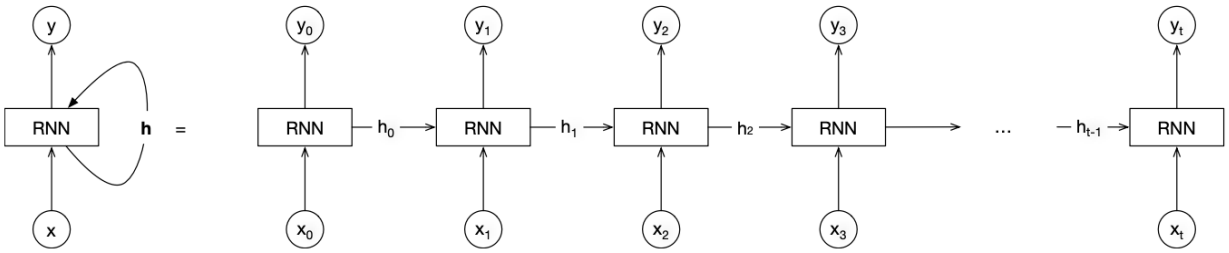

Figure 7: RNN Unrolled

- At each time step, the RNN receives an input and combines it with its current hidden state to produce an output and an updated hidden state.
- This updated hidden state then serves as the "memory" for the next time step.
- An RNN can be visualized as multiple copies of the same network, each passing information to its successor, forming a chain-like structure.

Inputs to an RNN

An RNN takes two primary inputs at each time step:

1. Current Input (x_t): The input data at the current time step. This could be a word in a sentence, a sensor reading, or any element of a sequence.
2. Previous Hidden State (h_{t-1}): The hidden state from the previous time step, which contains information about the past elements of the sequence. In the first time step, when there is no previous hidden state (no memory), it is initialized to some default value, often a vector of zeros.

Processing Steps

- The RNN block takes in the first input (x_1) along with the initial hidden state (h_0) and produces an output (y_1) and a hidden state (h_1)
- To process the second element in the sequence, the same RNN block takes in the hidden state from the previous timestep (h_1) and the input at the current timestep (x_2) to produce the output at the second timestep (y_2) and a new hidden state (h_2).
- This process continues until we reach the end of the sequence.

- After processing the entire sequence, we will have all the outputs at each timestep and the final hidden state.

What Happens to the Inputs

1. **Combination:** The current input (x_t) and the previous hidden state (h_{t-1}) are combined. This is typically done using weight matrices (W_{hh} and W_{xh}) and an activation function (i.e. tanh).

2. **Hidden State Update:** The combined information is used to update the hidden state:

$$h_t = \tanh(W_{hh} * h_{t-1} + W_{xh} * x_t)$$

3. **Output Generation:** The current hidden state (h_t) is used to produce an output (y_t):

$$y_t = W_{hy} * h_t$$

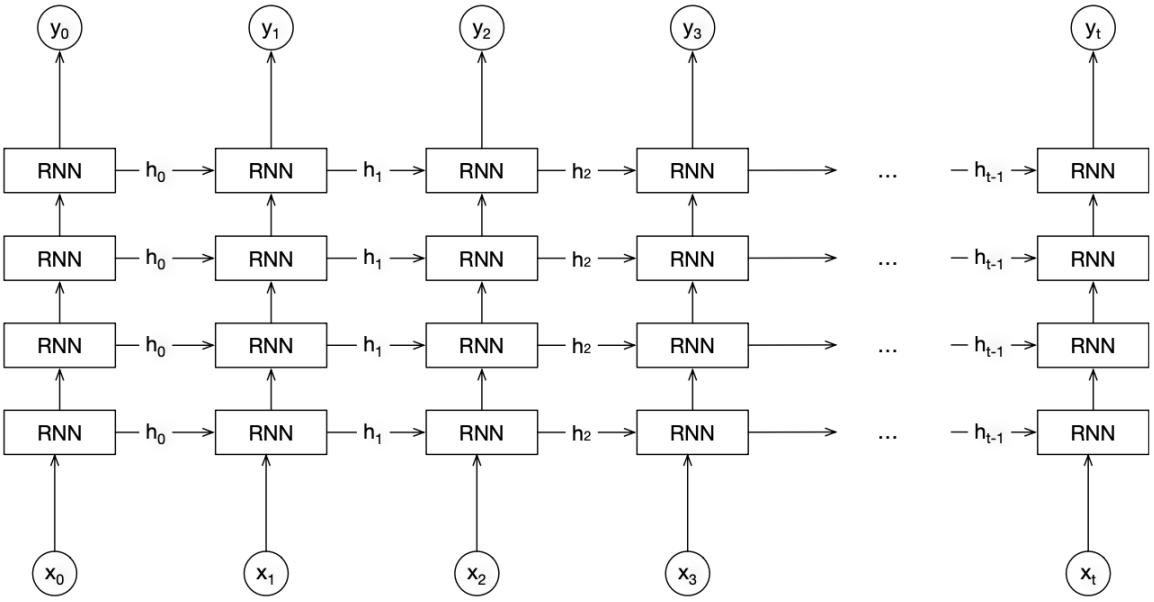

Figure 8: Possible RNN Structure

Structure of an RNN

- **Repeating Module:** The basic structure of an RNN consists of a repeating module (a neural network layer) that processes each element of the sequence.

- **Layers:** RNNs can be stacked to create deeper networks.

- **Bidirectional RNNs:** Process the input sequence in both forward and backward directions, concatenating the hidden states to capture information from both past and future contexts.

RNN Training

RNNs are trained using a special version of backpropagation called Backpropagation Through Time (BPTT) to find the optimal weights and biases in each update in RNN.

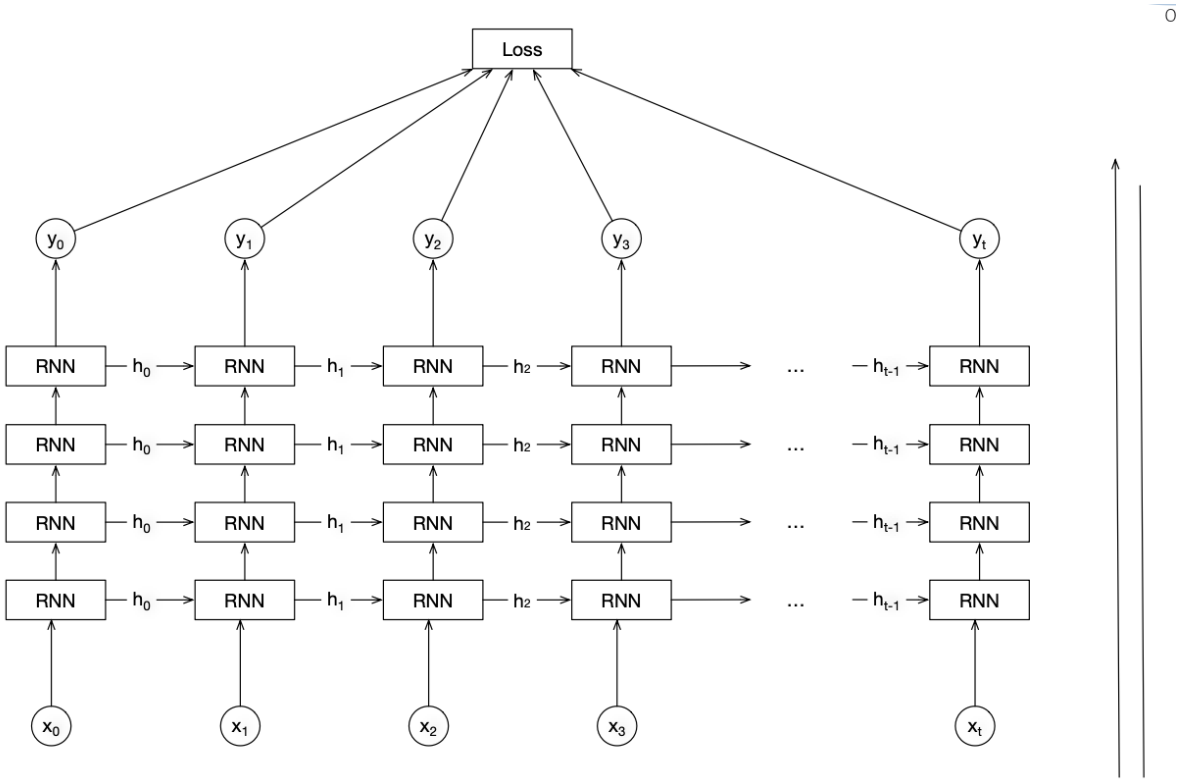

Figure 9: RNN Training

1. **Forward pass:** The basic idea of the forward pass in RNN is to calculate the values of hidden states in each time step and to compute the loss
2. **Backward pass:** Propagating the loss backward through entire sequence to compute the gradient and adjust the weights.

Limitations and Problems

Vanishing/Exploding Gradients:

$$h_t = \tanh(W(\tanh(W(\dots \tanh(W(h_0, x_1))) \dots)))$$

During training, due to repeated multiplications with W, the gradients can either vanish (become very small) or explode (become very large), making it difficult for the network to learn long-term dependencies.

Long Short-Term Memory (LSTMs)

The LSTM introduces a "memory cell" serving as long-term memory, in addition to the hidden state memory of classical RNNs.

LSTM Architecture

$$\begin{pmatrix} i_t \\ f_t \\ o_t \\ \tilde{c}_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$c_t = f_{t-1} \odot c_{t-1} + i_{t-1} \odot g_{t-1}$

$h_t = o_{t-1} \odot \tanh(c_t)$

LSTM has 4 gates to control the cell state. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!" Let the input to the LSTM at time t be X_t, and the hidden state from the previous timestep be H_{t-1}. Now, there are three gates that process information:

1. **Input Gate** (I_t): Decides how much information to read from the current input and previous hidden state.

$$I_t = \sigma(W_{xi}X_t + W_{hi}H_{t-1} + b_i)$$

2. **Forget Gate** (F_t): Decides how much information to forget from long-term memory.

$$F_t = \sigma(W_{xf}X_t + W_{hf}H_{t-1} + b_f)$$

3. **Output Gate** (O_t): Decides how much of the current cell state should be used to create the current hidden state, which is the output of the cell.

$$O_t = \sigma(W_{xo}X_t + W_{ho}H_{t-1} + b_o)$$

4. **Candidate Cell State** ($\tilde{C}_t$): The LSTM cell calculates a candidate cell state using another gate, but this time with tanh activation.

$$\tilde{C}_t = \tanh(W_{xc}X_t + W_{hc}H_{t-1} + b_c)$$

Where W_{xi}, W_{hi}, W_{xf}, W_{hf}, W_{xo}, W_{xc}, $W_{hc}$and W_{ho} are learnable weight parameters, and b_i, b_f, b_c and b_o are learnable bias parameters.

Memory Update

The cell state (long-term memory) is updated as follows:

$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t$

Here, $\odot$ represents element-wise multiplication. The forget gate decides how much information from the previous timestep to carry forward, and the input gate decides how much of the current candidate cell state will be written into long-term memory.

Finally, the hidden state is updated:

$H_t = O_t \odot \tanh(C_t)$

Training Process

1. **Forward Pass:** Input sequences are fed into the LSTM network. The gates (I_t, F_t, O_t), candidate cell state ($\tilde{C}_t$), cell state (C_t), and hidden state (H_t) are computed iteratively for each time step followed by loss function calculation.

2. **Backpropagation Through Time (BPTT):** The gradients of the loss for the network's parameters are calculated using BPTT. This involves propagating the error backward through time, and unrolling the LSTM cell over the sequence length. The network's parameters (weights and biases) are updated using an optimization algorithm (e.g., Adam, SGD) to minimize the loss function.

LSTMs solve the vanishing/exploding gradients problem as (C_t) is only multiplied element-wise and no repeated matrix multiplication with W occurs.

sLSTM (Scalar LSTM)

The core idea of sLSTM is to improve the control over the memory cell by enhancing the gating mechanism and introducing new forms of memory mixing.

1. **Exponential Gating:** Replaces the standard sigmoid gates in LSTM with exponential gates like $e^{\tilde{i}_t}$, followed by normalization to ensure values remain manageable and stable because, with appropriate normalization and stabilization techniques, these gates allow more flexible control over what information is stored or forgotten, making the model more adaptable and effective.

2. **New Memory Mixing:** sLSTM can manage several memory cells and heads, which allows it to mix different types of information within each head. This improves the model's ability to handle complex tasks but avoids mixing information between different heads to maintain clarity and stability. sLSTM is not parallelizable due to its memory mixing (hidden-to-hidden connections).

3. **Optimization:** To address this, fast CUDA implementations have been developed, using GPU memory optimizations at the register level.

4. **Constant Error Carousel (Cell State Update):** The fundamental memory update still uses the constant error carousel principle from LSTM:

   $c_t = f_t c_{t-1} + i_t z_t$

   However, the way f_t and i_t are calculated is modified with exponential gating....

mLSTM (Matrix LSTM)

1. **Matrix Memory:** The mLSTM replaces the scalar memory cell state with a matrix memory structure (C). The input information is represented as key-value pairs, where the keys are used to update the memory matrix, and the values are stored in the matrix. The query is used to retrieve information from the memory matrix.
2. **Covariance Update Rule:** Inspired by Bidirectional Associative Memories (BAMs), mLSTM uses a covariance update rule for efficient storage and retrieval of key-value pairs in the matrix memory.
3. **Parallel Processing:** The mLSTM abandons memory mixing (i.e., recurrent hidden-hidden connections), which enables parallel processing during training (parallel processing on GPUs) While this improves memory capacity, it increases computational cost due to complex matrix operations. This is similar to technologies like FlashAttention and GLA, which optimize processing speed.

Limitations of LSTMs

Despite their success, LSTMs have three major limitations:

1. **Lack of Parallelizability:** One of the main drawbacks of LSTMs is that they rely on sequential processing, where each time step depends on the previous one. Transformers have addressed this by using a self-attention mechanism, allowing them to process all time steps in parallel.
2. **Limited Storage Capacity:** LSTMs store information in scalar cell states, causing difficulty in handling rare events. Example: Rare Token Prediction in language models. LSTMs perform poorly on infrequent words in Wikitext-103. xLSTM improves this by using matrix memory instead of scalar memory.
3. **Inability to Revise Storage Decisions:** Example: Nearest Neighbor Search Problem LSTMs struggle to update stored values when a better match appears later in a sequence. The new xLSTM model addresses this issue using exponential gating.

Extended Long Short-Term Memory (xLSTM)

- xLSTM is an advanced variant of the traditional LSTM architecture.
- To overcome these issues, Extended Long Short-Term Memory (xLSTM) introduces two key improvements:
  - Exponential Gating – Improves information storage and retrieval.
  - Novel Memory Structures – Introduces sLSTM (scalar memory) and mLSTM (matrix memory).

Exponential Gating

Components of Exponential Gating

Exponential gating includes three gates coming from LSTM and Normalization and Stabilization.

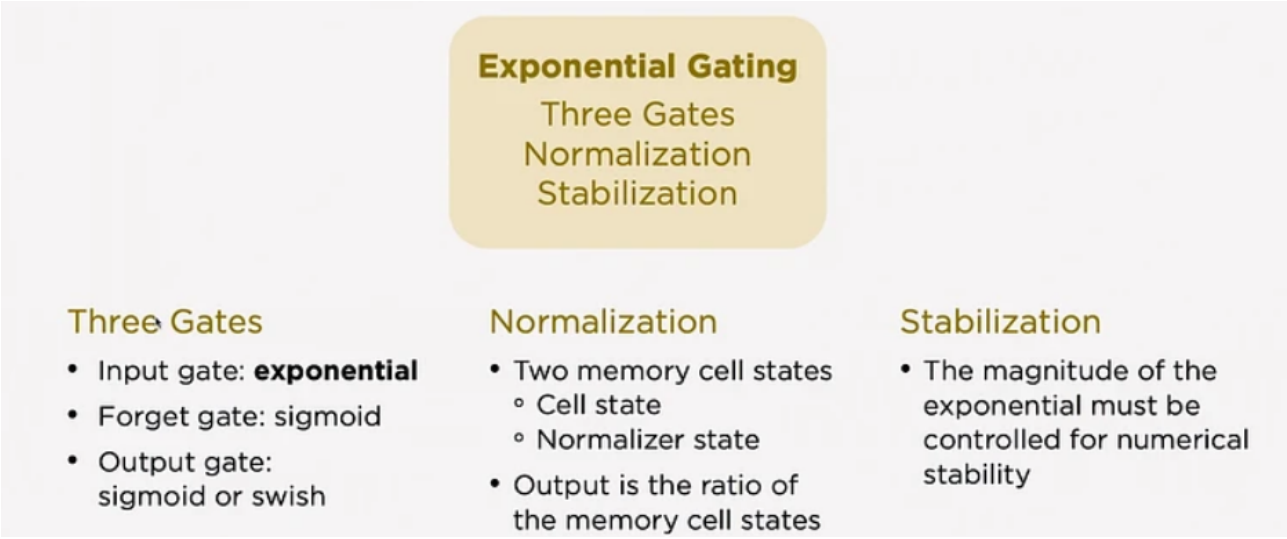

Figure 10: Exponential gating components

Like a standard LSTM, xLSTM maintains a cell state and hidden state but introduces optimizations in the gating mechanism. As depicted in Figure 11, LSTM is extended to exponential gating and applied to the input gate.

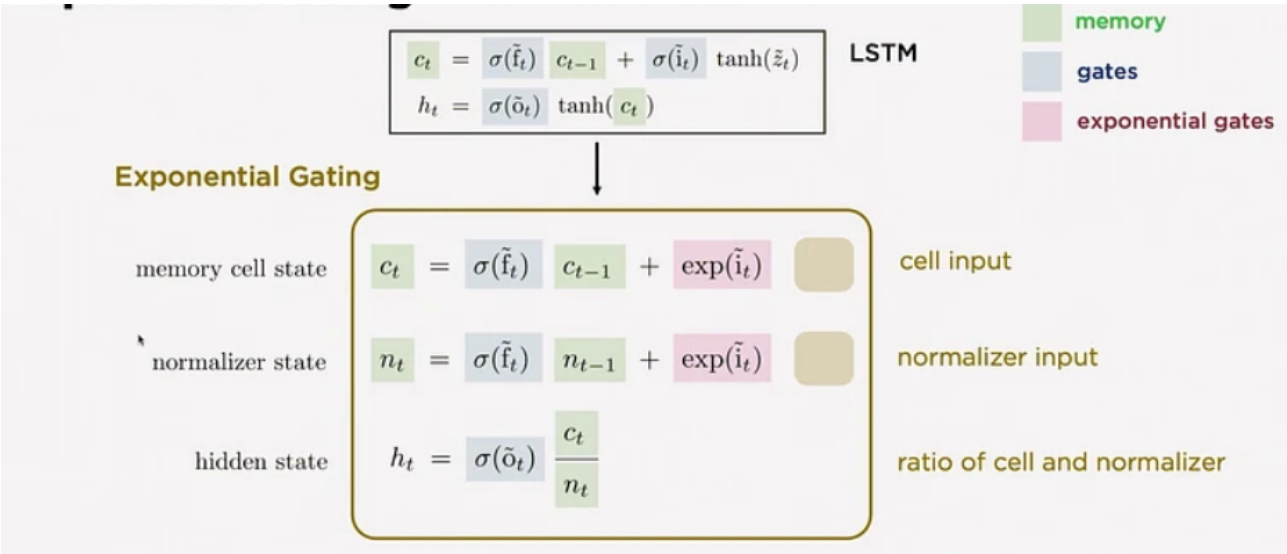

Figure 11: xLSTM core equations

Normalization in Exponential Gating

Normalization ensures that values remain within a **bounded range**, preventing unstable updates. Exponential gating introduces a **soft normalization technique** using:

$$\tilde{h}_t = \frac{h_t}{1 + |h_t|} \tag{1}$$

where:

- h_t is the **hidden state**.
- $\tilde{h}_t$ is the **normalized value**.

Effect:

- This prevents **extreme values** and keeps activations **smooth**.
- Helps in avoiding **gradient explosions**.

Stabilization in exponential gating

Exponential gating introduces stabilization by modifying how gate activations respond to changes in the hidden state.

Why is Stabilization Needed?

In traditional LSTMs:

- Gradients can explode if weights grow too large.
- Gradients can vanish if they shrink too much (especially in long sequences).
- Standard activation functions (sigmoid/tanh) can saturate, making learning slow.

A key stabilization mechanism in exponential gating is defined as:

$$g_t = \exp(-\alpha |h_t|) \tag{2}$$

where:

- g_t is the **stabilized gate activation**.
- h_t is the **hidden state** at time t.
- α is a **stability factor** controlling how fast stabilization occurs.

How It Works:

- **If** $|h_t|$ is **large**, the exponential decay ensures that g_t **shrinks**, preventing unstable updates.
- **If** $|h_t|$ is **small**, g_t remains **large**, ensuring that small signals are not ignored.

This mechanism prevents **sudden jumps in gradients** and ensures **smooth updates** over time.

Two New Memory Cells with Exponential Gating

xLSTM introduces two novel memory cells, sLSTM and mLSTM, both leveraging exponential gating for improved long-term memory retention. This dual-memory approach enables better feature extraction, reducing information loss in deep networks while maintaining computational efficiency. Both models outperform traditional LSTMs in tasks requiring robust temporal learning, such as ECG forecasting, speech recognition, and time-series prediction

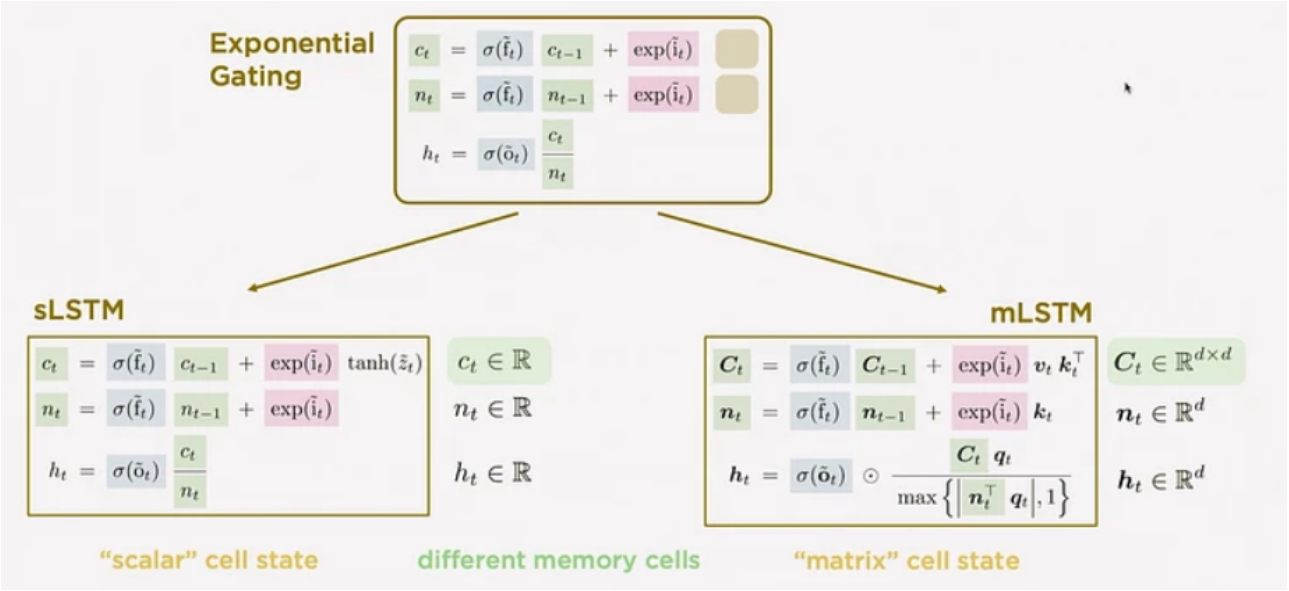

Figure 12: Memory cells in xLSTM

xLSTM Architecture

- xLSTM combines sLSTM and mLSTM into xLSTM blocks.
- These blocks can be residually stacked to form deep xLSTM architectures.
- This residual stacking improves training stability and performance

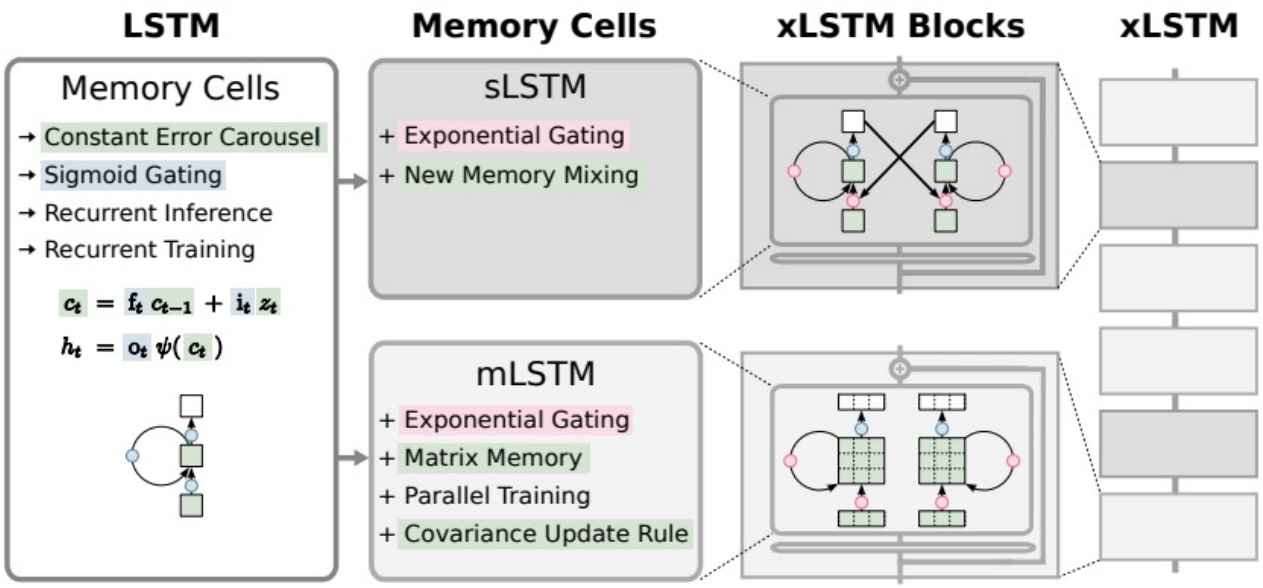

Figure 13: xLSTM Architecture

The xLSTM family (Beck et al., 2024). From left to right:

1. The original LSTM memory cell with constant error carousel and gating.
2. New sLSTM and mLSTM memory cells that introduce exponential gating. sLSTM offers a new memory mixing technique. mLSTM is fully parallelizable with a novel matrix memory cell state and new covariance update rule.

3. mLSTM and sLSTM in residual blocks yield xLSTM blocks.
4. Stacked xLSTM blocks give an xLSTM architecture.

xLSTM Limitation

The following are key limitations and challenges of xLSTM architecture:

1. **Memory Mixing in sLSTM Prevents Parallelization:**
   - The memory mixing structure in sLSTM hinders its ability to parallelize efficiently.
   - Its CUDA kernel is 1.5x slower than the parallel mLSTM version.
2. **CUDA Kernels for mLSTM Are Not Optimized:**
   - The CUDA implementation of mLSTM is currently 4 times slower than alternative approaches like FlashAttention or Mamba's Scan.
   - An optimized FlashAttention-like approach could improve performance.
3. **Matrix Memory in mLSTM Is Computationally Expensive:**
   - Despite the high computational cost, parallelization only marginally reduces wall-clock time.
4. **Forget Gate Initialization Must Be Carefully Handled:**
   - Proper initialization of the forget gate is critical to avoid convergence issues during training.
5. **Limited Memory Capacity ($d \times d$):**
   - The memory capacity is limited, but manageable for sequences up to 16k time steps.
6. **Neither Architecture Nor Hyperparameters Are Optimized:**
   - The current implementation lacks architectural refinements and has not been optimized with carefully fine-tuned hyperparameters.

xLSTMTime : Long-term Time Series Forecasting With xLSTM

Challenges with Existing Models (Alharthi & Mahmood, 2024)

Transformer Models

The use of Transformer models in time-series forecasting has gained significant attention, but they come with certain challenges:

- High computational complexity, particularly for long sequences, leading to increased resource consumption.
- Difficulty in capturing non-linear temporal dynamics inherent in many time-series datasets.
- Challenges in maintaining order sensitivity, which is crucial for time-series data.

- Susceptibility to noise, especially in volatile domains like financial forecasting.

Success of Simpler Models

Despite the challenges faced by Transformer models, simpler architectures have shown competitive performance:

- Models such as LTSF-Linear and ELM have outperformed transformers in specific forecasting tasks, demonstrating the efficacy of simpler approaches.
- These models also exhibit competitive performance while incurring fewer computational costs.

xLSTMTime Model

The xLSTMTime model is an advanced architecture built on the traditional xLSTM (Exponential Long Short-Term Memory) framework, specifically designed to enhance time series forecasting tasks by considering both trend and seasonal components present in the data.

Goal

The primary objectives of the xLSTMTime model are:

- To improve the accuracy of predictions in time series forecasting tasks.
- To provide an efficient and accurate forecasting model that balances high performance with computational efficiency, especially in domains that deal with complex, noisy, and volatile data.

Improving LSTMs

The xLSTMTime model enhances the traditional LSTM architecture by introducing the following key improvements:

- **Exponential Gating:** This mechanism is employed to better capture long-term dependencies, which is crucial for forecasting tasks that involve long-term patterns.
- **Revised Memory Structure:** The model incorporates a revised memory structure, which scales effectively for more complex and long-term forecasting tasks.

xLSTM Architecture - Data Processing Pipeline

Time Series Decomposition

The xLSTMTime model employs a decomposition technique to split the input time series data into two key components:

Trend Component The trend component captures the long-term patterns and tendencies in the data. This represents the underlying direction or movement over time, ignoring short-term fluctuations.

Seasonal Component The seasonal component captures the periodic fluctuations that occur at regular intervals. These fluctuations may be due to factors such as daily, weekly, or yearly cycles, which are inherent in many time series datasets.

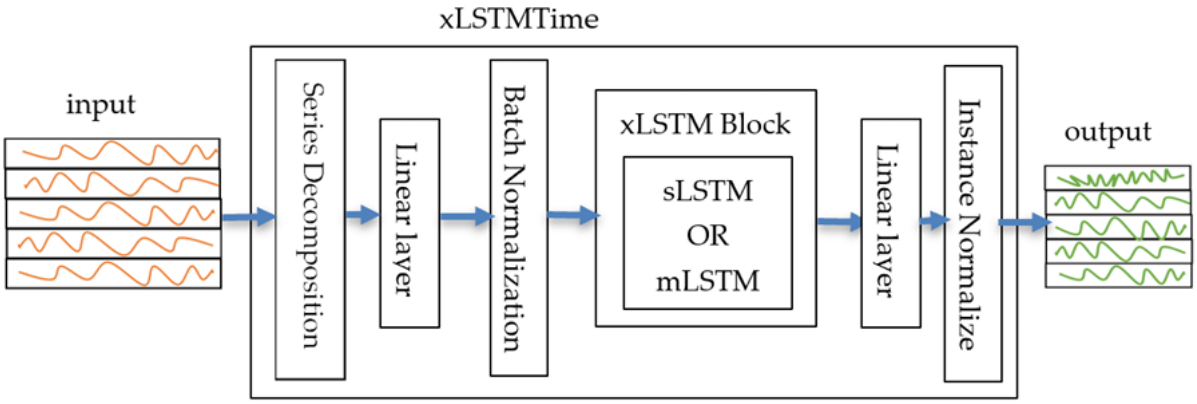

Figure 14: Architecture of xLSTMTime

$$\boldsymbol{x}_{trend} = AveragePool(Padding(\boldsymbol{x}))$$

$$\boldsymbol{x}_{seasonal} = \boldsymbol{x} - \boldsymbol{x}_{trend}$$

Batch Normalization

Batch Normalization is a technique used to stabilize the learning process and improve performance during training.

Purpose

The primary purpose of Batch Normalization is to stabilize and accelerate the training of deep neural networks. It helps ensure better performance by reducing issues related to the internal covariate shift and allows for higher learning rates during training.

How It Works

Batch Normalization normalizes the activations of each layer in the network. The normalization process ensures that each feature has zero mean and unit variance across the batch, thus preventing the network from struggling with varying input distributions.

Formally, for a given mini-batch with activations $x_1, x_2, \ldots, x_n$, the normalized activation $\hat{x}$ is computed as:

$$\hat{x} = \frac{x - \mu}{\sigma}$$

Where: - x is the activation of the layer, - μ is the mean of the activations, - σ is the standard deviation of the activations.

After normalization, a scale γ and a shift β are applied to allow the network to learn the optimal transformation for each feature:

$$y = \gamma\hat{x} + \beta$$

Benefits

Batch Normalization offers several advantages:

- **Higher Learning Rates:** By stabilizing the training process, Batch Normalization allows for higher learning rates, leading to faster convergence.
- **Faster Training:** Since Batch Normalization helps the network learn more efficiently, the training process is faster overall.
- **Reduces the Need for Dropout or Strict Initialization:** Batch Normalization reduces the reliance on techniques like Dropout or strict weight initialization, making training more stable without these regularization methods.

xLSTM Block – The Core of the Model

After the decomposition of the time series data into trend and seasonal components, the data undergoes further processing using the xLSTM block. This block, which incorporates both sLSTM and mLSTM, is designed to better learn the temporal dependencies within the data, improving the forecasting accuracy.

Components of the xLSTM Block

- **sLSTM (Scalar Memory)** Manages long-term dependencies with exponential gating. Focuses on controlling memory for historical data.
  - Suitable for smaller datasets (e.g., ETTm1, ILI).
- **mLSTM (Matrix Memory)** Utilizes matrix memory for larger datasets. Enhances memory storage and retrieval for complex patterns.
  - Handles larger datasets (e.g., Electricity, Traffic).

Linear Layer

After the data has undergone processing through the xLSTM block, it passes through a final **linear transformation layer** to adjust the output. This layer performs a linear transformation on the processed data, helping the model make the final predictions by mapping the learned features to the desired output space.

The linear transformation is represented as:

$$y = W \cdot x + b$$

Where:

- y is the output of the linear layer.
- W is the weight matrix, which is learned during training.
- x is the input to the linear layer (the output from the xLSTM block).

- b is the bias term.

This layer is crucial for mapping the learned features from the xLSTM block to the final prediction, which can be used for time series forecasting or other tasks depending on the model's objective.

Instance Normalizer

The **Instance Normalizer** is applied independently to each channel of the time series data. This technique ensures that each component (or channel) of the data has a mean of 0 and a variance of 1, improving the stability of the model during training.

The process of instance normalization can be expressed as:

$$\hat{x}_t = \frac{x_t - \mu_t}{\sigma_t}$$

Where:

- x_t is the raw input at time step t for a specific channel.
- μ_t is the mean of the input over the current batch (for each channel).
- σ_t is the standard deviation of the input over the current batch (for each channel).
- $\hat{x}_t$ is the normalized output for that time step.

By normalizing each channel independ

Setting Up xLSTMTime: A Comprehensive Guide for Beginners

Download and Setup

Download the zip file `xLSTMTime-main.zip` from Google Drive and extract it to the desired location.

Google Drive link: Google Drive link

This includes original code from the author of the xLSTM-Time model along with modifications required for the PTBXL dataset. The original Github repository for xLSTM-Time can be found here: xLSTM-Time-Sourcecode

Virtual Environment Setup (Optional)

If you want to create a virtual environment, follow these steps:

1. Open Terminal (you can use vscode terminal or your machine terminal).
2. Navigate to `xLSTMTime-main` using the `cd` command.
3. Run `python -m venv .venv`.
4. Run `source venv/bin/activate` to activate the newly created virtual environment.
5. Inside the virtual environment, run `pip install -r requirements.txt`.

Package Installation (Without Virtual Environment)

Run `pip install -r requirements.txt` to install the required packages in the global Python environment.

Running the Code

Try to run `main.py`.

Code Structure and Process Flow

Folder: all_six_datasets

This folder contains datasets used by the author, in `.csv` format. New datasets should be added here inside folders with names defined in `DSETS` in `datautils.py`. Ensure there is a `.csv` dataset file inside dedicated folders.

File: data_preprocessing.py

This file reads ECG signal files from the `PTB-XL-Dataset` folder and creates a `.csv` file that can be fed to the model.

File: main.py

This file contains crucial parameters to set, which are passed to `args` to the model.

Parameters/Hyperparameters

- `-dset`: Dataset name from the list `'DSETS'` defined in `datautils.py`.
- `-context_points`: Sequence length (window size) of time-series data fed into the model. Larger values might be better for long-range dependencies. Smaller values might be sufficient and faster to train for shorter patterns. A good starting point would be to test values equal to your data's primary periodicity.
- `-target_points`: Forecast horizon - the number of time steps the model predicts into the future. In `Dataset_PTBXL`, this corresponds to `pred_len`. Integer, default: 96.
- `-scaler`: Scaling method for input data. String, default: `'standard'`. Options: `'standard'` (StandardScaler), `'minmax'` (MinMaxScaler), or `None` (no scaling).
- `-features`: Specifies which features to use from the dataset - `'M'` (Multivariate) or `'S'` (Single/Univariate).
- `-batch_size`: Number of samples processed in each training iteration. Larger batch sizes can lead to faster training but might require more memory.
- `-num_workers`: Number of subprocesses to use for data loading. Integer, default: 1. Set to the number of CPU cores.

- `-patch_len`: Length of each patch. Integer, default: 12.
- `-stride`: Stride between patches. Integer, default: 12. If `stride == patch_len`, patches are non-overlapping. If `stride < patch_len`, patches overlap.
- `-revin`: Flag to use RevIN for normalization. Integer, default: 1 (True).
- `-n_epochs`: Number of training epochs.
- `-is_train`: Flag to indicate whether to train the model (1) or run in testing mode (0).
- `-n_layers`: Number of layers in the model.
- `-d_model`: Dimension of the transformer layers (embedding dimension).
- `-dropout`: Dropout rate.
- `-head_dropout`: Head dropout rate. Used to prevent overfitting on heads.

Functions

- `find_lr()`: This function sets up the training components (data, model, loss, callbacks) and uses a learning rate finder to determine a suitable learning rate. Returns suggested learning rate.
- `train_func()`: Trains the model using the suggested learning rate, configuring `get_dls(args)` in `datautils.py` and a class in `src/data/pred_dataset.py`.
- `test_func()`: Saves models in `saved_models/-dset` and stores predicted, target values in a list named `out`. Returns `out`.
- `plot_features_actual_vs_predicted()`: Plots actual vs predicted values for different features, based on the defined feature range.

File: datautils.py

This file contains the list `DSETS` with names of datasets. It prepares the dataset for training a deep learning model using PyTorch.

Function: get_dls()

This function has a condition for each dataset defined in `DSETS` to check the current dataset defined in `main.py`. Each condition block forwards parameters to the Dataset loading class defined in `pred_dataset.py`. The `get_dls` function loads and validates the dataset by checking the shape of a sample.

Key Components:

- *Imports:* Uses PyTorch, NumPy, Pandas, and custom modules (`DataLoaders`, `Dataset_PTBXL`).
- *Parameters:* `context_points`, `target_points`, `batch_size`, `num_workers`, `features`.
- *Dataset Loading:* The dataset should be in the path specified in `root_path` and is loaded from `patient.csv`.

Usage:

- The `Params` class sets the configuration values.
- The `get_dls` function returns a `DataLoaders` object containing the training data.
- After loading, it prints the shape of the first training sample.

Customization:

- Modify `root_path` and `data_path` for your dataset location.
- Adjust parameters for batch size, features, and time points.

File: src/data/pred_dataset.py

This file contains classes for datasets to handle training, testing, validation splits, defining features in the dataset, and the size/shape of the dataset. `Dataset_PTBXL` and `Dataset_longterm` classes are added to handle these datasets.

Class: Dataset_PTBXL

This class is a custom implementation of a dataset used for time-series forecasting with ECG data. It inherits from `torch.utils.data.Dataset`.

Constructor (`__init__` method): Initializes the dataset, sets configuration options, and prepares the data for loading.

Key Arguments:

- `root_path`: The directory where the dataset is stored.
- `split`: The dataset split type (train, val, or test).
- `size`: Defines the sequence length, label length, and prediction length.
- `features`: Specifies the features to use (`'M'` or `'S'`).
- `target`: The target column.
- `scale`: Whether to normalize (scale) the data.
- `timeenc`: Defines how time encoding works.
- `freq`: Frequency of time steps.
- `time_col_name`: The column name containing timestamps.
- `use_time_features`: Whether to use time-related features.
- `train_split` & `test_split`: Ratios for splitting the data.

Reading and Processing the Data (`__read_data__` method):

- *Data Loading:* Reads data from a CSV file at `data_path` within `root_path`.
- *Scaling:* Normalizes the dataset using `StandardScaler` if `scale=True`.
- *Feature Selection:* Selects columns based on the `features` argument.
- *Timestamps:* Converts the `time_col_name` to a datetime format.

Data Slicing: Splits the dataset into windows of data: `seq_len`, `label_len`, `pred_len`.

Indexing (`__getitem__` method): Retrieves input (`seq_x`) and output (`seq_y`) sequences.

Length of Dataset (`__len__` method): Returns the number of possible samples in the dataset.

Inverse Transformation (`inverse_transform` method): Reverses the scaling operation.

File: xlstm1/slstm/cell.py

sLSTM needs GPU for faster processing, in this file it can be defined. Options are CUDA or Vanilla. No mention of MacBook's MPS framework.

Folder: saved_models

Trained models can be found in this folder, which can be tested by setting the `-is_train` parameter in `main.py` to 0.

Transformers

Transformer Attention Concepts

The foundation of modern transformer architectures lies in the self-attention mechanism, which enables models to dynamically weigh the importance of different elements in a sequence. Here are the core concepts:

Architecture of Transformer

Input Embedding and Positional Encoding

1. **Input Embedding**: Converts input tokens (e.g., words or time-series data points) into vector representations.
2. **Positional Encoding**: Adds information about token positions since Transformers lack inherent sequential processing.

Encoder

The encoder consists of N identical layers. Each layer has:

1. **Multi-Head Attention**: Captures relationships between different tokens in the input.
2. **Add & Norm (Residual Connection + Layer Norm)**: Ensures stable training and helps gradients flow efficiently.

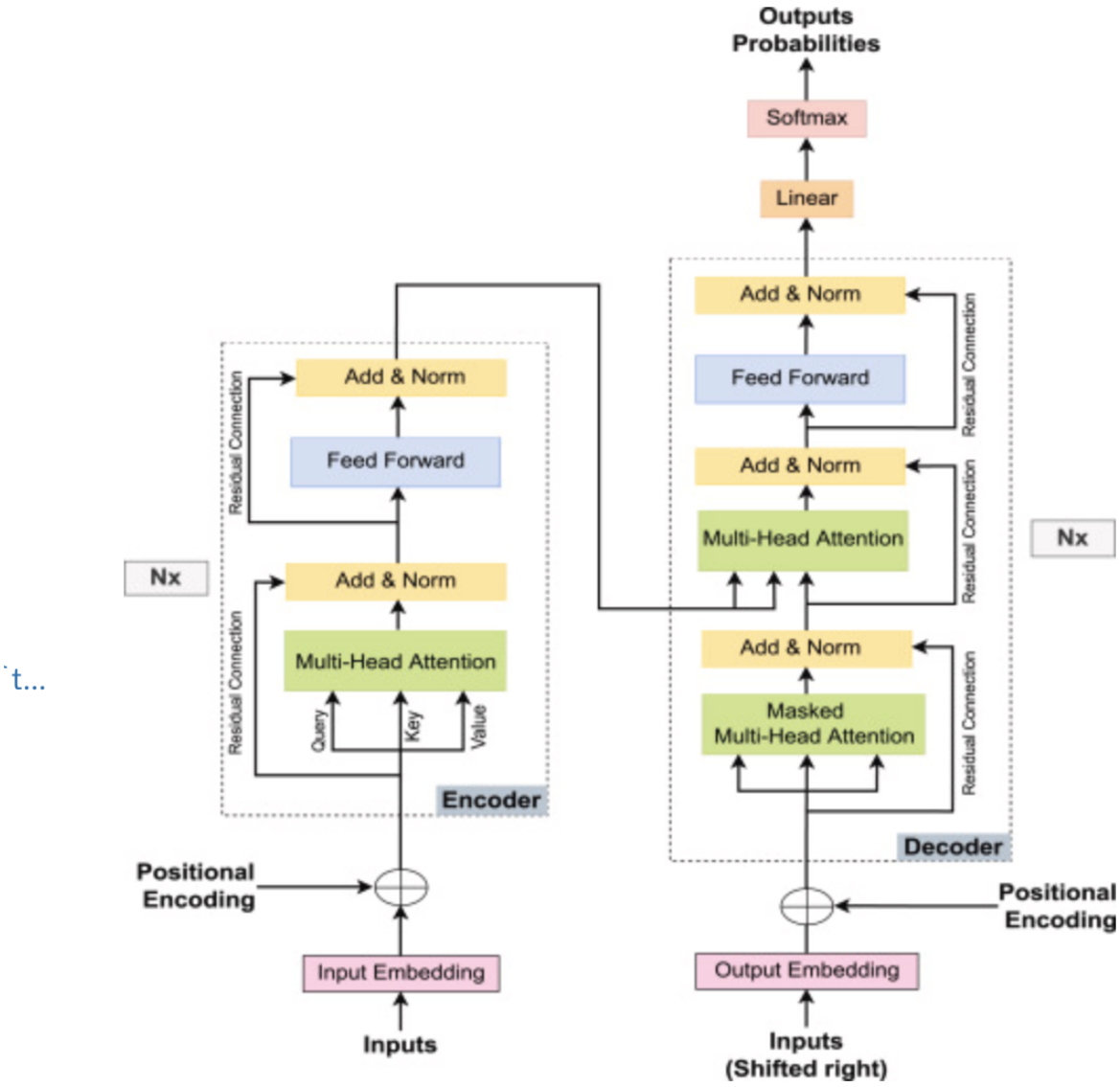

Figure 15: Architecture of Transformer [4]

3. **Feed-Forward Network (FFN)**: Fully connected layers that transform features for better representation.

Decoder

The decoder also has N identical layers but includes additional components:

1. **Masked Multi-Head Attention**: Ensures predictions only depend on past tokens (prevents peeking at future data)
2. **Multi-Head Attention (with Encoder Output)**: This layer attends to encoder outputs, helping the decoder generate context-aware predictions.
3. **Add & Norm + Feed-Forward Network**: Similar to the encoder, normalizing and refining features.

Output Generation

1. **Linear Layer**: Maps the decoder's final representation to logits (raw scores).

2. **Softmax Layer**: Converts logits into probabilities for the final prediction.

Transformers for Time Series Analysis

Time series data presents unique challenges: long-term dependencies, multivariate inputs, and dynamic/static covariates. Transformers address these through:

1. **Handles Long-Term Dependencies** – Unlike LSTMs, transformers use self-attention to capture long-range patterns efficiently.
2. **Parallel Processing** – Unlike RNNs, transformers process entire sequences simultaneously, leading to faster training.
3. **Feature Selection & Interpretability** – Models like Temporal Fusion Transformer (TFT) provide variable importance analysis for better insights.
4. **Multivariate Time-Series Forecasting** – Can process multiple correlated signals (e.g., sales, weather, and stock prices) effectively.
5. **Handles Missing Data** – Unlike LSTMs, transformers do not require strictly sequential data, making them robust for real-world datasets.

Temporal Fusion Transformer (TFT)

The Temporal Fusion Transformer (TFT) is a deep learning model for time-series forecasting that combines LSTMs and self-attention to capture both short-term patterns and long-term dependencies. It is designed to be interpretable, allowing feature importance analysis while handling multivariate, multi-horizon forecasting efficiently. TFT uses variable selection, gated residual networks (GRNs), and multi-head attention to improve performance and explainability.

Architecture of TFT

Key Components of TFT

1. **Variable Selection**
   - **Purpose:** Identifies important input features dynamically at each time step.
   - **How?** Uses gated mechanisms to select relevant static, past, and known future variables.
2. **LSTM Encoder & Decoder**
   - **Purpose:** Captures temporal dependencies in the input data.
   - **How?** Uses separate LSTM encoders and decoders to process historical and future inputs.
3. **Static Covariates Encoders**
   - **Purpose:** Encodes static metadata (e.g., store ID, product category) that remains constant.

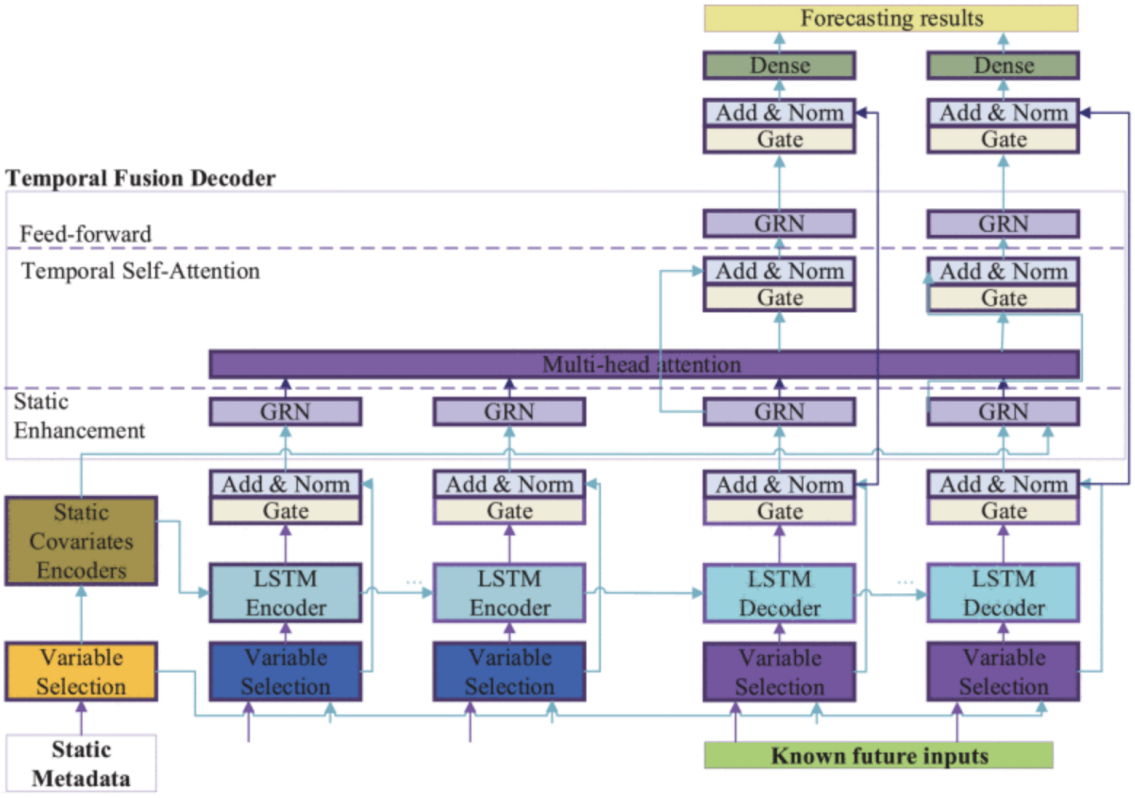

Figure 16: Architecture of TFT [5]

- **How?** Uses a Gated Residual Network (GRN) to learn representations for static features.

4. **Multi-Head Attention**

   - **Purpose:** Captures dependencies across multiple time steps.
   - **How?** Applies multi-head attention to extract relevant patterns.

5. **Gated Residual Networks (GRNs)**

   - **Purpose:** Helps in non-linear feature transformations.
   - **How?** Uses skip connections, gating mechanisms, and normalization to stabilize learning.

6. **Dense Layer & Forecasting Results**

   - **Purpose:** Produces the final predictions for future time steps.
   - **How?** Uses fully connected (dense) layers followed by normalization and gating.

TFT for ECG Forecasting

Time-series forecasting plays a crucial role in ECG analysis. In the section below, we'll explore the Temporal Fusion Transformer (TFT) for forecasting ECG Lead 0 using the PBL-XL dataset.

TFT was chosen because it outperforms traditional sequence models in long-range forecasting and offers the following advantages:

1. **Handles multiple features effectively:** Uses a Variable Selection Layer to identify important ECG signal components dynamically.
2. **Captures both short and long-term dependencies:** Incorporates an LSTM encoder-decoder for local trends and Multi-Head Attention for global patterns.
3. **Provides interpretability:** Features like Gated Residual Networks (GRNs) and Temporal Self-Attention help understand which time steps and variables influence predictions the most.

Implementation Steps

Data Preprocessing

Dataset: The PTB-XL dataset, which contains ECG signals and metadata, is used for time-series forecasting.

Libraries Used: `wfdb, pandas, numpy, ast, pytorch_forecasting, torch, sklearn`

Loading Data:

- ECG signals are loaded using the `wfdb` library.
- The function `load_raw_data(df, sampling_rate, path)` reads ECG waveforms.
- Data is processed to create an individual row for each time index, and the processed data is saved in CSV format.
- The function `process_patient_data(...)` extracts metadata such as patient ID, age, sex, and ECG waveform data.

Data Preparation and Feature Engineering

Frequency Reduction:

- Original ECG signals recorded at 1000 Hz are downsampled to 500 Hz by removing alternate rows.
- This reduces computational complexity while preserving essential characteristics.

Data Splitting:

- The dataset is split into training (80%), validation (10%), and test (10%) sets.

Normalization:

- ECG data is normalized using `TorchNormalizer` with robust scaling to handle outliers.

Variable Selection:

- Static features: Age, sex.
- Time-varying features: ECG leads, time index.

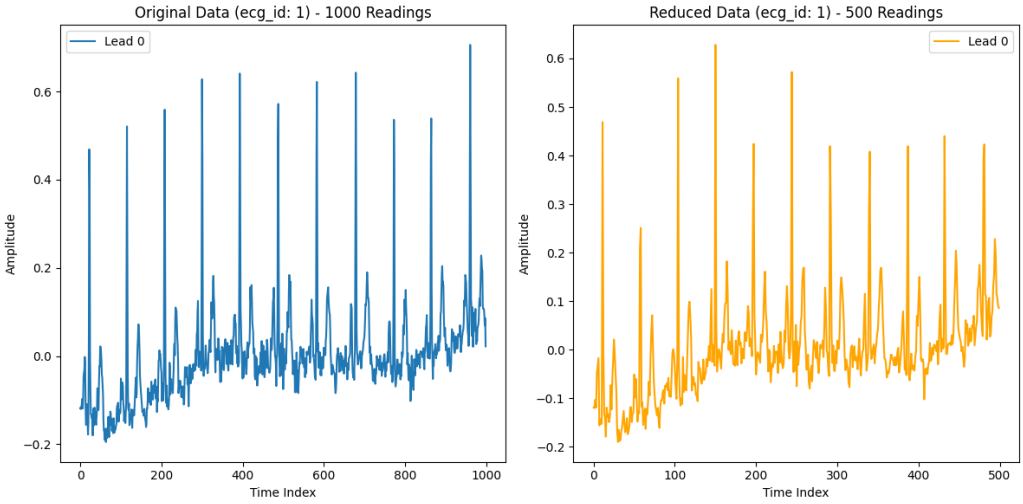

Figure 17: Downsampled data

Time Series Dataset Creation:

- `TimeSeriesDataSet` objects are created for training, validation, and testing.
- Encoder length = 300, Prediction length = 200 for `TFT` model.

DataLoader Configuration:

- Batch sizes: 80 (training), 80 (validation), 5 (testing).

Model Architecture:

`TFT` model is built using `pytorch_forecasting`.

LSTM Layers:

- 4 LSTM layers capture temporal dependencies in ECG data.

Attention Mechanism:

- Multi-head attention with head size = 256.

Hidden Layers:

- Hidden size = 256 for complex feature extraction.

Dropout:

- Dropout rate = 0.1 to prevent overfitting.

Loss Function:

- Quantile Loss with quantiles [0.05, 0.5, 0.95].

Training Configuration

- **Early Stopping:** Patience = 10 epochs, monitors validation loss.

- **Gradient Clipping:** Value = 1.0 for stability.
- **Training Epochs:** Maximum 50 epochs with early stopping.

Model Output

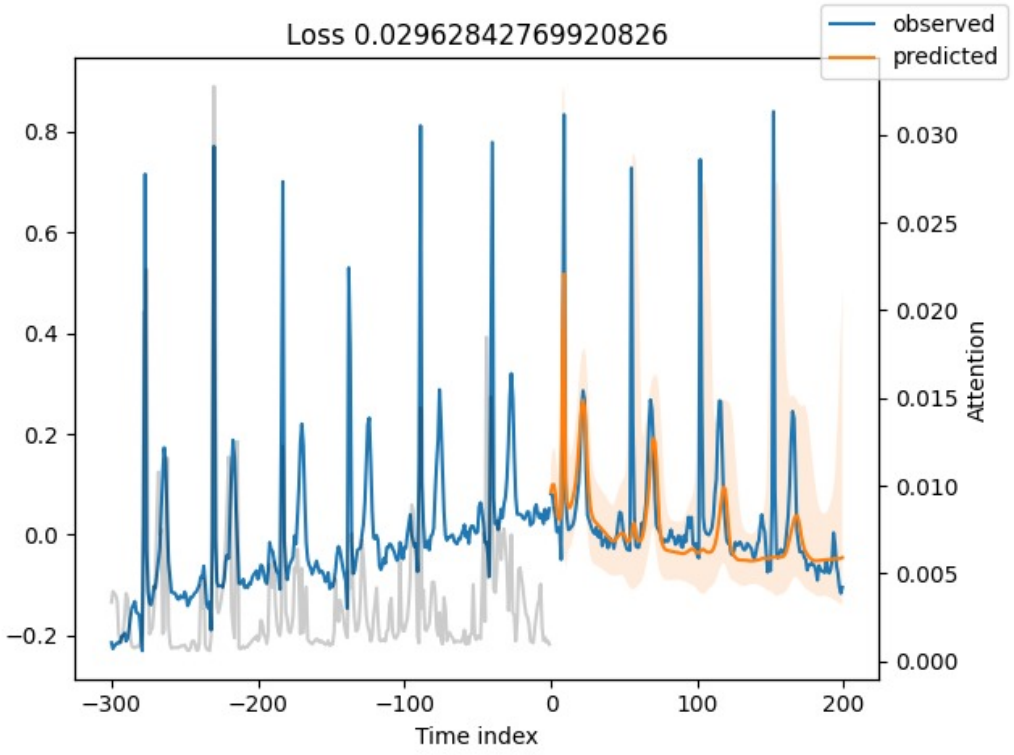

Figure 18: Predictions vs Actual values

Performance Metrics

Evaluation Metrics:

- Mean Absolute Error (MAE): 0.0544
- Root Mean Squared Error (RMSE): 0.1142
- Symmetric Mean Absolute Percentage Error (SMAPE): 1.0534
- Quantile Loss (0.05, 0.5, 0.95): 0.0299

Interpretation of Results:

- Low MAE and RMSE indicate good model performance.
- SMAPE shows reasonable accuracy across varying scales.
- Quantile loss values demonstrate robustness.

Interpretation from Model

Attention Score:

- A line graph represents attention values over time.

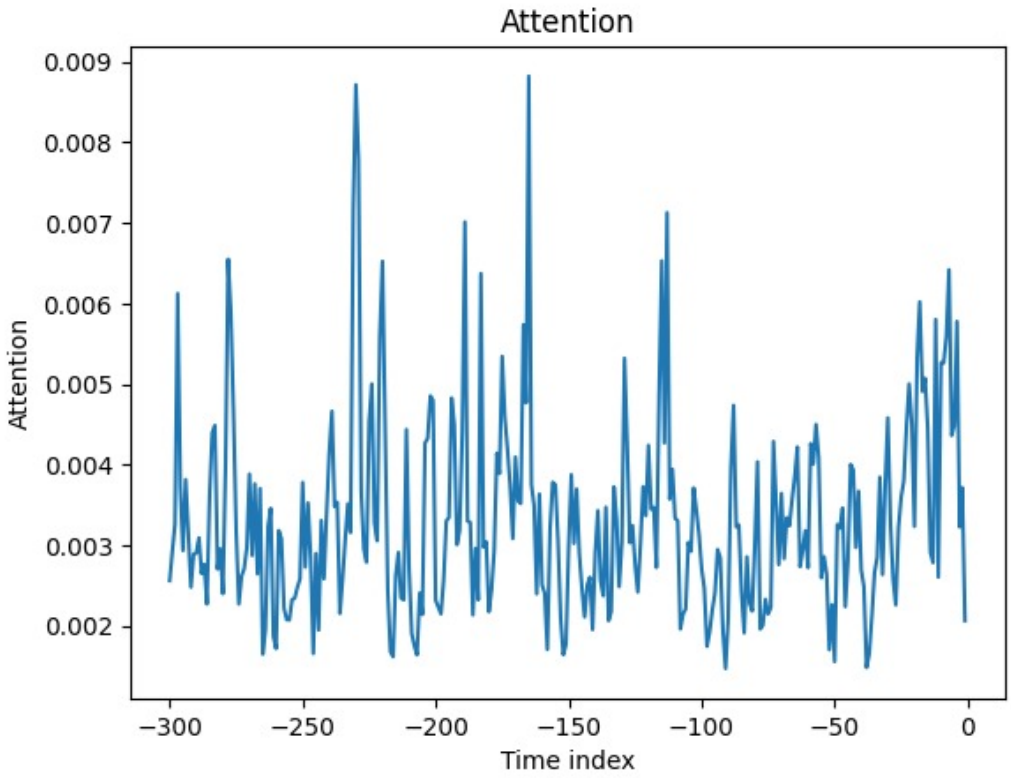

Figure 19: Attention Weights

Static Variable Importance:

- Age is highly significant for ECG pattern prediction.
- Sex has lower importance ($\approx 10\%$).

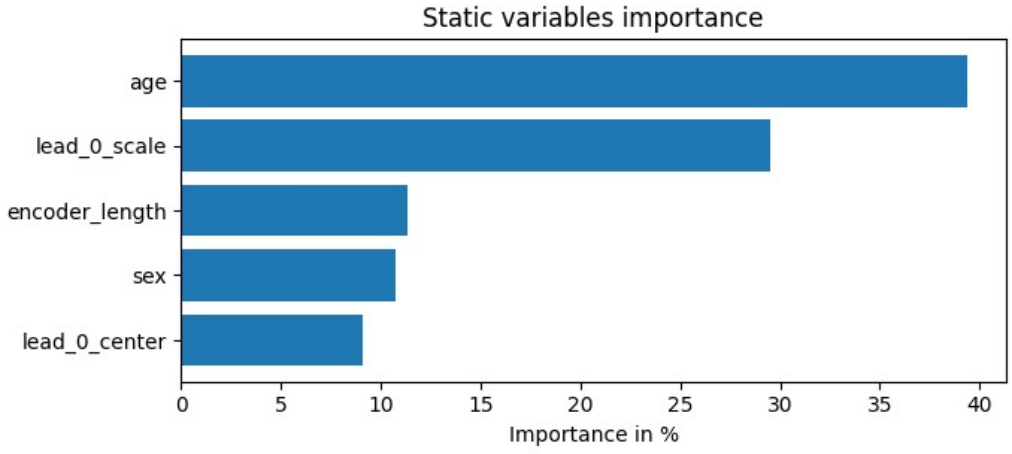

Figure 20: Static variables importance

Encoder Variable Importance:

- `Lead_0` is the most critical ECG feature ($\approx 18\%$).
- Other leads (e.g., `Lead_7, Lead_3, Lead_1, Lead_2`) contribute significantly.

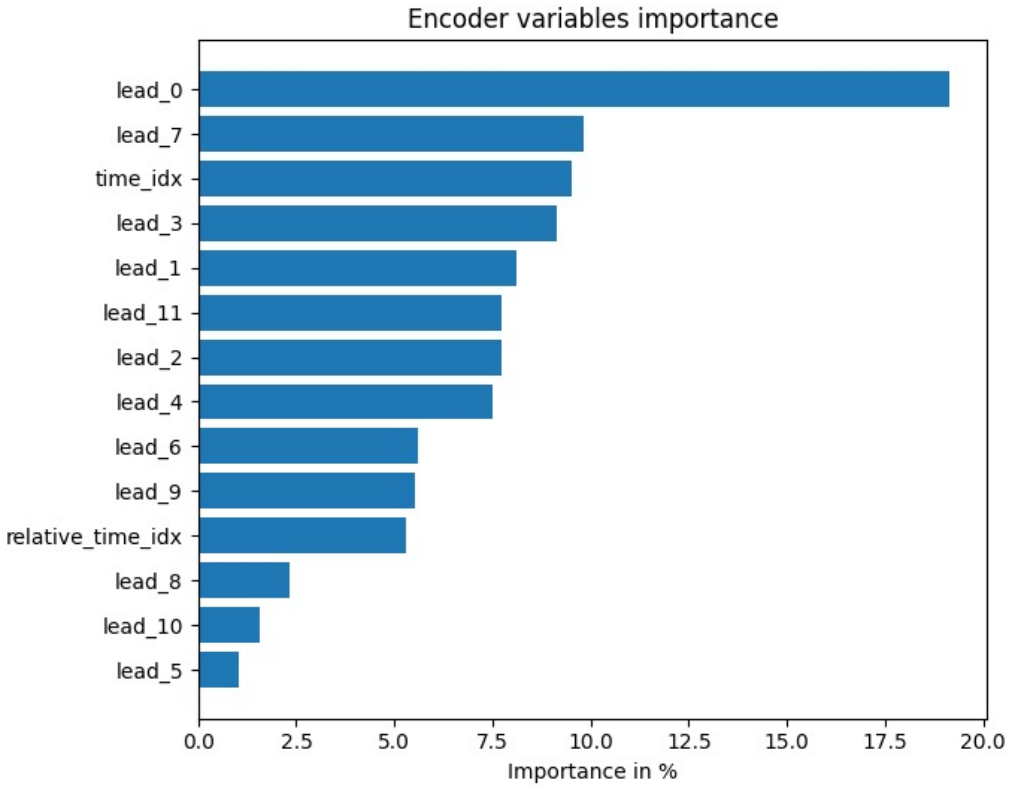

Figure 21: Encoder variables importance

Instruction manual to run TFT ECG Forecast

Download and Setup

First, download the file `TFT_ECG_Forecast.ipynb` from GitHub. Once downloaded, extract it to your desired location.

Requirements

Before running the code, ensure you have installed all dependencies specified in `requirements.txt`. Install them using the command below:

```
pip install -r requirements.txt
```

Dataset Setup

Modify the dataset path in the code to point to the correct location of your ECG data.

How to Run

Once the dataset path is updated, execute the Jupyter Notebook:

TFT_ECG_Forecast.ipynb

Conclusion

Transformers and Temporal Fusion Transformers (TFT) have emerged as powerful tools for time series analysis, offering significant advantages over traditional models like LSTMs and RNNs. The self-attention mechanism in Transformers enables efficient handling of long-term dependencies and parallel processing, making them well-suited for complex time series data. TFT, in particular, combines the strengths of LSTMs and self-attention to provide interpretable, multivariate, and multi-horizon forecasting capabilities.

The application of TFT in ECG forecasting, as demonstrated with the PTB-XL dataset, highlights its ability to handle multivariate time series data with missing values and dynamic/static covariates. The model's performance, evaluated using metrics such as MAE, RMSE, SMAPE, and quantile loss, shows strong predictive accuracy and robustness. Additionally, the interpretability features of TFT, such as attention scores and variable importance analysis, provide valuable insights into the factors influencing predictions, such as the significance o

References

[1] M. Joseph, *Modern Time Series Forecasting with Python: Explore Industry-Ready Time Series Forecasting Using Modern Machine Learning and Deep Learning*. Packt Publishing Ltd, 2022.

[2] Talent500, "Time series forecasting with long short-term memory (lstm) networks in tensorflow," 2023. [Online]. Available: https://talent500.com/blog/time-series-forecasting-with-long-short-term-memory-lstm-networks-in-tensorflow/

[3] A. Pandey, P. Mannepalli, M. Gupta, and et al., "A deep learning-based hybrid cnn-lstm model for location-aware web service recommendation," *Neural Process Letters*, vol. 56, p. 234, 2024.

[4] S. Islam, H. Elmekki, A. Elsebai, J. Bentahar, N. Drawel, G. Rjoub, and W. Pedrycz, "A comprehensive survey on applications of transformers for deep learning tasks," *Expert Systems with Applications*, vol. 241, p. 122666, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417423031688

[5] H. Liao and K. K. Radhakrishnan, "Short-term load forecasting with temporal fusion transformers for power distribution networks," in *2022 IEEE Sustainable Power and Energy Conference (iSPEC)*, 2022, pp. 1–5.