Revolutionizing Time Series Data Preprocessing with a Novel Cycling Layer in Self-Attention Mechanisms

JIYAN CHEN

A THESIS SUBMITTED TO

THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF ARTS

GRADUATE PROGRAM IN

INFORMATION SYSTEMS & TECHNOLOGY

YORK UNIVERSITY

TORONTO, ONTARIO

April 2024

Abstract

This thesis presents a novel method for improving time series data preprocessing by incorporating a cycling layer into self-attention mechanisms. Traditional techniques often struggle to capture the cyclical nature of time series data, impacting predictive model accuracy. By integrating a cycling layer, this thesis aims to enhance the ability of models to recognize and utilize cyclical patterns within datasets, exemplified by the Jena Climate dataset from the Max Planck Institute for Biogeochemistry. Empirical results demonstrate that the proposed method not only improves the accuracy of forecasts but also increases model fitting speed compared to conventional approaches. This thesis contributes to the advancement of time series analysis by offering a more effective preprocessing technique.

Acknowledgements

Completing this master's thesis has been one of the most challenging yet rewarding experiences of my academic journey. It would not have been possible without the support, guidance, and encouragement of many individuals, to whom I owe my heartfelt gratitude.

First and foremost, I extend my deepest thanks to my supervisor, for her invaluable mentorship, patience, and expert advice. Her insights and constructive criticism were instrumental in shaping this research, and her belief in my capabilities inspired me to strive for excellence. I am also profoundly grateful to other professors, whose expertise and feedback significantly contributed to the quality of this thesis. The opportunity to work under the guidance of such distinguished scholars has been a privilege that has enriched my learning experience immensely.

I would like to acknowledge my peers and colleagues at York University, for their camaraderie and the intellectually stimulating environment they provided. Their friendship and support have made this journey more enjoyable and memorable. Lastly, I would like to extend my gratitude to my family members for their help and mental support along the journey. This thesis is not just a reflection of my efforts but a testament to the collective support and guidance of all those mentioned above. To everyone who has been a part of this journey, thank you from the bottom of my heart.

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1 Background

Time series analysis is a fundamental aspect of various scientific disciplines, ranging from economics to engineering. At its core, a time series is a sequence of data points, typically consisting of successive measurements made over a time interval. These measurements can be taken at uniform time intervals or irregular intervals, such as daily stock market prices and the occurrence of natural disasters.

Historically, time series data has been used to track and predict phenomena like ocean tides, sunspot counts, and stock market indices, such as the Dow Jones Industrial Average. Among these phenomena, the essence of time series analysis lies in its ability to extract meaningful statistics and other characteristics from the data. Based on the correlation between previous observations and future values, time series data is important in predicting future phenomena. This forecasting is not just a mere extrapolation but is rooted in understanding intricate patterns, trends, and seasonality in the data. For instance, in the field of economics, time series analysis aids in discerning long-term trends from month-to-month variations, ensuring that economic policies are not misguided by short-term fluctuations [1].

Meanwhile, time series analysis has been widely used in various fields and has made important contributions to the orderliness of human productive activities. In the realm of economics and finance, it is used to predict stock prices and used as economic indicators. In engineering, it aids in signal processing and system control. In the natural sciences, it is employed for weather forecasting and earthquake prediction. The versatility of time series analysis is evident in its adaptability to both linear and non-linear models, as well as its applicability to both univariate and multivariate data.

However, there are still some challenges in time series analysis. First and foremost, the data can exhibit different patterns that are seasonal, cyclical, or even chaotic. Seasonal fluctuations, such as increased sales during specific seasons, can skew predictions if not accounted for. In general, cyclical patterns do not follow fixed time intervals, which makes them more difficult to predict, as in the case of economic booms and recessions. In addition, some of the data originally appear randomly, for example, weather patterns are random and chaotic in nature. However, these patterns are in turn controlled by deterministic laws, such as monsoons and ocean currents, which exacerbates the complexity of time series analysis in predicting weather patterns.

Furthermore, the non-stationarity of data can lead to unreliable forecasting results if it is not addressed. This non-stationarity poses significant challenges in modeling and forecasting. Moreover, the challenges of dealing with missing values, irregular sampling intervals, and the need for domain-specific expertise further complicate the process. To address these data challenges, various methodologies and techniques have

been developed. From autoregressive models to Fourier transforms, these tools aim to provide a clearer understanding of the data and enhance prediction accuracy.

Secondly, in addition to the analytical shortcomings inherent in the data, a number of factors from external sources may cause confusion in the data patterns, which exacerbates the difficulty of predicting the phenomenon. Therefore, these data must be rigorously preprocessed before they can be effectively analyzed.

Data preprocessing is the process of cleaning and transforming raw data before it can be used for machine learning, data mining, or any type of data analysis, as they can significantly affect the performance of subsequent data analysis or predictive modeling tasks. For instance, the missing values in time series data can lead to significant gaps in understanding patterns, while outliers can skew the results of predictive models. These issues emphasize the need for time series analysis which ensures that the data entering the analytical model are both accurate and representative of the underlying phenomenon.

The unreasonable data preprocessing methods can also expose different problems. While the intent is to refine the data, overzealous or inappropriate preprocessing methods can distort the data's inherent characteristics. For instance, overly aggressive smoothing techniques might erase short-term fluctuations that are critical to understanding certain patterns. Similarly, imputing missing values without understanding the reason behind their absence can introduce biases. In the area of time series analysis, the sequential nature of the data adds another layer of complexity. For real-world time series data exposing seasonality and trends, some techniques (e.g.,

differencing or decomposition) may not be always effective in separating or capturing these components. This is especially true when the data does not follow linear trends or multiple overlapping seasonal patterns. Another prevalent issue aspect in preprocessing time series data involves handling missing values and outliers. Incomplete or inconsistent data can mislead the analysis severely, and lead to inaccurate predictions. Existing methods for dealing with these issues (e.g., linear interpolation or mean substitution) may be too simple to adequately reflect the inherent complexity of time series data.

Another major gap is lack of automation in data preprocessing. Currently, many preprocessing steps are manual, which is not only time-consuming but also carries the risk of human error. Meanwhile, preprocessing must also avoid a common pitfall of the “one-size-fits-all" approach, which does not allow for full optimization of the data for the specific analysis or forecasting task at hand.

The advent of big data and machine learning has led to a surge in the development of advanced algorithms and models for time series analysis. Researchers used leveraged time-series datasets to forecast yields under varying climatic conditions, concluding that machine learning techniques combined with remote sensing data can effectively model yield predictions influenced by climate change. These modern approaches, combined with traditional methods, offer a comprehensive toolkit for researchers and practitioners alike.

This thesis emphasizes the importance of time series analysis due to its wide range of

potential applications and the complexities involved. Therefore, it is crucial to delve deeper into the techniques used in time series analysis, understand the hurdles encountered in this field, and investigate future opportunities for advancement. This will not only enhance the robustness of our research but also provide valuable insights for future studies in this domain.

1.2 Problem Definition

Time series data, with its sequential nature and inherent complexities, posed unique challenges for preprocessing. Traditional methods often fall short of capturing the intricate temporal dependencies and patterns within the data. The problem of capturing intricate temporal dependencies within data that has periodic features is a significant challenge in the field of data analysis, especially in the aspect of concerning time series and spatial-temporal data.

One of the primary difficulties is that most existing approaches tend to overlook the long-term dependencies inherent in the data. They either fail to consider the whole mechanism required to model these long-term dependency issues or use batch techniques that cut long sequences into many short ones, which does not effectively capture the dependencies between sequences, a problem referred to as sequence fragmentation [2].

Another issue is the shifting of long-term periodic dependency which is often ignored in current studies. For instance, the traffic data exhibited strong daily and weekly

periodicity, demonstrating that understanding and utilizing dependencies from previous periods can be beneficial for prediction tasks. Yet, many models do not adequately account for these dependencies [3].

1.3 Significance

This thesis addresses the challenge of capturing intricate temporal dependencies in periodic data, a task that holds substantial significance in various analytical and predictive contexts. The relevance of this issue is evidenced by a body of research emphasizing the limitations of existing models in accurately depicting these dependencies. In contrast, conventional approaches often overlook long-term dependencies, resulting in fragmented data sequences that fail to reflect the true temporal dynamics.

In response to these challenges, advancements have been made through the development of innovative neural network architectures. For instance, the introduction of the Spatial-Temporal Adaptive Module (STAM) within the Adaptive Hybrid Spatial-Temporal Graph Neural Network (AHSTGNN) represents a targeted effort to accommodate the complexities of spatial-temporal data. Such innovations are pivotal in advancing the field of temporal data analysis and in enhancing the predictive capabilities of models [4].

The application of these advanced models is shown in the improved performance of Bi-LSTM in modeling traffic data, which has demonstrated the ability to capture and utilize

periodic characteristics for prediction. This not only serves as a proof of concept but also sets a benchmark for the potential of similar models in other real-time decision-making scenarios [5].

The significance of improving temporal dependency modeling is also reflected in the broader impact on various disciplines. For example, in economics, the ability to predict market trends with greater accuracy can lead to more informed economic decisions. In the environmental sciences, better climate models can enhance our understanding and responses to environmental changes [6].

The goal of this thesis is to utilize weather coefficient data to capture periodic features. Addressing these challenges requires innovative techniques tailored specifically for time series data. This research proposes a pioneering approach to meet the challenges of preprocessing time series data. The key of the approach involves replacing traditional positional encoding techniques with a personalized Cycling technique built into the Self-Attention Mechanisms.

1.4 List of Contributions

The significance of this research lies in its innovative approach to preprocess time series data, a domain that has been challenged by the intricacies of temporal dependencies for a long time. The key contributions are discussed in the following paragraphs:

Firstly, this thesis contributes to the time efficiency of data processing and speed of regression in time series analysis. Time efficiency is paramount in data preprocessing,

especially when dealing with vast datasets. The novel Cycling Layers not only improves accuracy but also maintains a speed advantage, especially in ergodic situations. This speed advantage can be observed as faster regression which allows the model to achieve desired loss scores in significantly less time compared to traditional methods. The experiments based on time series dataset have shown that the Cycling Layer achieves its objectives faster than the positional layer in Self-Attention Mechanisms. The improved performance marks a step forward in time series preprocessing.

Secondly, this thesis improves the accuracy and reduces the error of data prediction in time series analysis. Traditional positional encoding frequently fails to meet expectations of capturing the nuances of time series data. By introducing the Cycling Layers, this research has observed a significant improvement when compared to the positional layer of Self-Attention Mechanisms. This is exhibited when using the Mean Squared Error (MSE) loss function, a standard metric to assess performance. A lower MSE score indicates a more accurate representation of the data. The experiments have shown that Cycling Layers consistently yield a lower error rate throughout the autoencoder process, suggesting that the notable advancement over existing autoencoder technologies. The result from this thesis displays an outperformance when processing time series data, such as weather change, but further research is required to investigate the best approach for other types of data.

Finally, this thesis highlights the adaptability and customizability of time series analysis. One of the standout features of the Cycling Layers in Self-Attention Mechanisms is

their adaptability. Traditional preprocessing technologies often offer a one-size-fits-all solution, which may not always be optimal for specific analytical tasks. In contrast, the Cycling Layers can be tailored to capture regular features of different time series data by adjusting the cycling period. In this experiment, to ensure that the model accurately captures daily fluctuations, the cycle period was set to one day to analyze daily energy consumption patterns. Therefore, if each piece of data represents an interval of minutes, this thesis sets the cycling period 60*24 to represent one day. Furthermore, the architecture of the Cycling Self-attention Layer allows for customization, enabling users to modify and train models specific to their objectives. This flexibility ensures that the model is not only accurate but also adaptable to various analytical challenges.

In conclusion, this research intends to pave the way for more accurate, efficient, and customizable solutions in the domain by addressing the challenges in time series data preprocessing with the introduction of the Cycling Layers in Self-Attention Mechanisms.

1.5 Thesis Outline

This thesis is structured to offer a thorough understanding of the challenges, methodologies, and contributions in the realm of time series data preprocessing using Cycling Layers in Self-Attention Mechanisms. The subsequent chapters are organized as follows:

Chapter 2 explores a literature review, focusing on time series analysis, traditional

preprocessing techniques, and the evolution of Self-Attention Mechanisms; Chapter 3 discusses the intricacies of time series data, its unique characteristics and patterns, and the challenges it presents; Chapter 4 provides a detailed introduction to the proposed Cycling Layers methodology, its integration into Self-Attention Mechanisms, and further incorporation into Autoencoders; Chapter 5 presents and analyzes the experimental results, comparing the proposed methodology with existing techniques, and evaluates its strengths and limitations; Chapter 6 concludes the thesis, summarizing the key findings, discussing broader implications, and suggesting avenues for future research and improvements.

2. Literature Review

2.1 Traditional Time Series Data Processing

The examination of time series data processing boasts a long-standing and continuously developing history, adapting to the ever-changing challenges it encounters. Time series analysis is crucial because it impacts many areas [7]. Hence, the time series data process leads experts from different fields to address its challenges. In earlier times, researchers relied on digital signal processing and random processes to analyze time series data. These methods helped them get a clear picture of the data, revealing its patterns, shifts, and unique points.

- ***Statistical Approach*** The statistical approach to processing time series data has evolved into a rich field of study within machine learning, reflecting the ubiquity and importance of temporal data across various domains. As the survey by Dama and Sunquest [6] demonstrated that the field is not only theoretically rich but also immensely practical with a diverse array of applications that underscore its significance. Deeks [8] discussed the importance of meta-analysis in combining results from multiple studies. It emphasized the need for careful consideration of study designs, biases, and heterogeneity. Jujjuru [9] reviewed various statistical

models commonly used in time series analysis, such as autoregressive models and integrated moving average models, discussing their strengths and weaknesses. Dancker [10] gave an overview of five statistical methods for time series forecasting: Autoregression (AR), Moving Average (MA), ARMA, ARIMA, and SARIMA.

- Digital Signal Processing Approach Time series data processing has witnessed significant evolution over the decades, especially in the field of digital signal processing. The progression of methodologies and techniques can be broadly categorized based on the shifts in research focus and the challenges addressed.

Before 1980, the initial forays into time series processing were grounded in linear models, with a strong emphasis on Orthogonal, Stationary, and Gaussian situations. Techniques laid the groundwork for digital signal processing in time series analysis, such as the Adaptive Filter, encompassing LMS (Least Mean Squares), LSL (Least Squares Lattice), and FTF (Fast Transversal Filter), the classical AutoRegressive Integrated Moving Average (ARIMA) model, particularly its AR component. Meanwhile, the use of the Fourier transform for trend decomposition was a hallmark of this era, providing a robust tool to analyze and predict time series data [11-13].

The 1980s marked a paradigm shift from linear to non-linear models. The focus expanded to array processing [14], which allowed for a more complex and nuanced analysis of time series data. This decade was characterized by the exploration of

non-linear dynamics in time series, recognizing that real-world data often exhibited non-linear patterns that linear models could not capture effectively.

The latter half of the 1990s saw a surge in the adoption of Time-Frequency analysis techniques [15]. Time-frequency analysis is a method used when dealing with signals, like sound waves or radio waves. In those waves, many parts of signals have changed as time passes. This analysis helps the researcher understand how the different pitches or speeds within a signal change over time. This is especially crucial for non-stationary data, where the statistical properties change over time. Techniques like the Wavelet Transform became popular during this period, offering a multi-resolution analysis of time series data and capturing both time and frequency domain information.

The early 2000s marked the prominence of Bayesian methods [16] in time series analysis. Bayesian techniques, grounded in probability theory, provide a framework to update the probability estimate for a hypothesis as more evidence or information becomes available. In the context of time series, Bayesian methods offer a way to incorporate prior knowledge and uncertainty into the models, leading to more robust and adaptive analyses. These methods are particularly effective in handling non-Gaussian distributions and offer a principled approach to model selection and uncertainty quantification.

- Random Processes Approach Many applications have been foundational in time series forecasting and have their roots in the study of random processes. Therefore,

random processes have been a cornerstone in the study of time series data processing for many decades. These processes are conceptualized as collections of random variables indexed by time. Two primary categories emerged in this process, discrete-time and continuous-time random processes. The former is defined at specific time points, while the latter spans continuous intervals.

Overall, time series analysis has primarily utilized statistical and econometric models, such as Autoregressive Integrated Moving Average (ARIMA) models and Exponential Smoothing [16]. These models are effective in modeling linear dependencies and are extensively used for forecasting. Meanwhile, a large number of studies have been conducted on data preprocessing, which have been applied to real-life cases. Most of the studies have focused on difficulties arising from the nature of the low-quality data and the objectives themselves, which are known as anomalies that may manifest themselves as noise [17], missing values [18], or useless variables [19]. Cortés-Ibáñez et al. [20] proposed a novel preprocessing methodology and analysis in a typical case. His methodology included noise reduction and time series smoothing using the central average, along with the selection of crucial features based on the importance assigned by the utilized regression algorithms. In this case, the Gradient Boosting [21] and Random Forest [22] algorithms were selected as validation algorithms because they shared some common features that matched the nature of the problem. Additionally, these two algorithms provide a measure of feature importance, aiding in the selection of the most relevant attributes. In summary, most

of the early research on time series data processing and preprocessing primarily revolved around improving traditional independent mathematical models for specific cases.

2.2 Advanced Time Series Data Processing

Traditional models for time series analysis, while foundational, have begun to reveal their limitations as the intricacy and sheer volume of time series data surge. The linear assumptions and fixed structures of these models struggle to capture the nuances of modern datasets, especially with the rise of IoT devices, high-frequency trading, and other sources generating vast amounts of sequential data.

Recognizing these challenges, the research community has started to pivot toward more flexible and adaptive machine-learning techniques. These techniques are better equipped to handle the non-linearity, high-dimensionality, and intricate temporal dependencies inherent in contemporary time series data.

- ***RNN Approach*** The 1980s marked a significant turning point with the advent of neural networks, which promised a more data-driven approach to time series analysis [23]. These networks, unlike their traditional counterparts, can learn intricate patterns from data without relying heavily on prior assumptions. Among the various neural network architectures, Recurrent Neural Networks (RNNs) have stood out for their ability to process sequences. They were designed to remember past information, making them particularly suited for time series data.

-LSTM Approach However, standard RNNs have their own set of challenges, primarily the vanishing and exploding gradient problems, which make them less effective at capturing long-term dependencies. This has led to the development of more advanced RNN variants. Long Short-Term Memory (LSTM) units, introduced by Hochreiter and Schmidhuber [24] in 1997, addressed these issues with specialized gating mechanisms. These gates control the flow of information, allowing LSTMs to remember or forget data over longer sequences. Following LSTMs, Gated Recurrent Units (GRUs) were proposed as a simpler alternative, retaining the efficiency of LSTMs but with fewer parameters, making them computationally more efficient [25]. Recently, the focus on the preprocessing of time series data has shifted towards the use of neural network relative technologies to replace the traditional models. Sagheer et al. [26] introduced a pre-trained LSTM-based stacked autoencoder (LSTM-SAE), which replaced random weight initialization in deep LSTM networks in an unsupervised manner. Kieu et al. [27] proposed solutions for outlier detection for time series data using ensembles of recurrent autoencoders, which built autoencoders using sparsely connected recurrent neural networks (S-RNNs). What makes these possible is that the classical autoencoder is a feedforward, fully connected neural network with identical numbers of neurons in the input and output layers, and significantly fewer neurons in the hidden layers [28].

Autoencoders, a type of artificial neural network, are designed for unsupervised learning of efficient coding. Their primary objective is to learn a representation

(encoding) for a set of data, typically for dimensionality reduction or denoising. The structural principle of the autoencoder is to reproduce its input as close as possible. This is achieved by forcing the data through a bottleneck that acted as a compression mechanism to minimize the number of neurons. The underlying principle is that by reducing the dimensionality in the hidden layers, the autoencoder is compelled to capture only the most salient and representative features of the data. This inherently means that it tends to ignore finer details, which often includes noise, outliers, or other non-essential information. As a result, the output of the autoencoder, after the decoding process, is a cleaner and denoised version of the input.

-CNNs Approach More recently, Convolutional Neural Networks (CNNs) have demonstrated effectiveness in time series data analysis [29]. WaveNet, a specific CNN structural, has shown superiority in forecasting financial time series. Additionally, combining CNN with LSTM and simple exponential smoothing has generated high-quality time series data that improves forecasting performance. They excel in recognizing local and global patterns in data and were often used in conjunction with LSTMs or GRUs for optimal performance. Some specific CNN models, such as S-CNN [30], are novel hybrid methods. It combines exponential smoothing with CNN, outperforming traditional models such as MLP and LSTM in forecasting accuracy. This approach uses an optimum smoothing factor and has been tested across multiple datasets to ensure consistency.

Interestingly, with the development of deep learning and neural networks, the

mentioned signal processing field also emphasizes the use of related technologies, designed to handle advanced techniques and methodologies to handle complex data structures and patterns.

Feedforward Neural Networks (FNN) are the most straightforward type of artificial neural network architecture, where data flows from the input layer to the output layer without looping back. Popular models include Multilayer Perceptrons (MLP), which consist of multiple layers of nodes in a directed graph [31]. Unlike feedforward neural networks, RNNs (Recurrent Neural Networks) have connections that loop back, making them well-suited for processing sequences of data. They can remember previous inputs, allowing them to exhibit dynamic temporal behavior [32]. Self-organizing neural networks represent the state-of-the-art in the analysis of time series data; these networks learn to classify input vectors based on their grouping in the input space. The most popular variant is the Kohonen network, which uses competitive learning to adjust weights [33].

2.3 Limitations and Challenges

The results of these models are not always satisfactory in capturing the features of time series data. They also present certain limitations and challenges during the processing phase.

The key tasks in time series data analysis include monitoring, forecasting, and anomaly detection. These tasks can significantly enhance an organization's ability to make

informed decisions, especially when executing in real-time. However, the processing and analysis of time series data in real-time pose unique challenges, requiring robust and scalable streaming frameworks to handle the data efficiently as it arrives. For other models, Nunes et al. [34] conducted a comparative analysis of two statistical methods, the seasonal autoregressive integrated moving average (ARIMA) and cyclical regression models. These two models analyzed Interrupted Time Series (ITS) data, which was related to mortality. Their findings indicated that the seasonal ARIMA method effectively produced non-autocorrelated residuals, particularly during weeks with increased deaths that contributed to influenza epidemics. Compared to the cyclical regression model, the seasonal ARIMA model demonstrates a superior performance with a lower residual mean square.

Consequently, this has led to a narrower 95% confidence interval in the estimates derived from the seasonal ARIMA model, enhancing its efficiency [35]. Wen et al. [36] pointed out an issue that the Vanilla Transformer model operated without making predefined assumptions about data patterns and characteristics. While it serves as a versatile and universal network capable of modeling long-range dependencies, this flexibility comes at a cost. Specifically, it requires a substantial amount of data for training. This extensive data requirement is crucial to enhance the model's generalization capabilities and to prevent overfitting [35].

In summary, the limitations and challenges in processing time series data primarily stem from the volume and continuous nature of the data. The ongoing evolution of

processing frameworks and the active debate between statistical and machine-learning approaches for analysis and forecasting underscore the dynamic and complex nature of this field.

2.4 Related Study for the Research Data

The Jena Climate dataset provided a comprehensive set of weather time series data from January 1, 2009, to December 31, 2016. The dataset is hosted on Kaggle and collected by the Max Planck Institute for Biogeochemistry in Jena, Germany. It includes 14 meteorological measurements such as air temperature, atmospheric pressure, humidity, and wind direction, recorded every 10 minutes. This extensive and granular dataset is highly valuable for climate studies and time series data analysis. The preprocessing of this data, crucial for efficient feature extraction and time prediction modeling, focuses on optimizing data dimensionality. Based on the collected data, various related research projects have been conducted.

The analysis of the Jena Climate dataset by Li [37] involved a detailed Exploratory Data Analysis (EDA) to understand the dataset's characteristics, followed by the application of an ARIMA model for time series forecasting. This approach predicts future climate conditions based on historical data, which helps to reveal patterns, trends, and correlations in climate variables such as temperature and humidity. The outcome of such an analysis is invaluable for understanding climate dynamics and could potentially contribute to climate change studies or weather forecasting efforts.

The "Jena Climate Prediction with LSTM" analyzed by Qin [38] and the "Daily Forecasting LSTM-FB Prophet" analyzed by Yacoub [39], both utilized advanced machine learning techniques for climate data forecasting. Qin's approach focused on employing Long Short-Term Memory (LSTM) networks. Relying on the strengths in dealing with long-term dependencies in the Jena climate dataset, this method can accurately predict variables such as temperature and humidity. On the other hand, Yacoub's method combined the deep learning capabilities of LSTMs with the robust, intuitive features of Facebook's Prophet tool. Aiming to enhance the precision of daily climate predictions. This dual-model methodology in Yacoub's analysis was particularly adept at capturing complex temporal patterns and adapting to seasonal variations and irregularities. In summary, both analyses emphasized the effectiveness of cooperation between traditional and modern forecasting techniques in predicting climatic conditions. This made a significant contribution in areas such as weather forecasting, environmental monitoring, and climate change research.

The 'Tensorflow 3 - RNN' analyzed by Shen [40], employed advanced Recurrent Neural Network (RNN) techniques within TensorFlow to analyze the Jena Climate dataset. This approach focused on leveraging RNNs to capture complex temporal relationships in climate data, such as temperature and humidity. The goal is to forecast future climate conditions or to accurately extract meaningful insights from the data's temporal patterns. It prioritizes model optimization, training, and evaluation to ensure predictions are both accurate and reliable.

The "Wrangling Concepts with Time Series Data" analyzed by Muhammad [41], focused on the meticulous preparation and exploration of the Jena Climate dataset, emphasizing data wrangling techniques crucial for time series analysis. This process included addressing challenges such as missing values, data normalization, and outlier handling. Exploratory data analysis was then conducted to reveal trends, seasonality, and patterns in climate variables. By leveraging Python libraries for data manipulation and visualization, the analysis aimed to establish a solid foundation for more complex time series forecasting, thereby providing insightful and reliable interpretations of climatic trends and behaviors.

3. Techniques

Machine learning has witnessed a paradigm shift with the advent of attention mechanisms, fundamentally transforming how models process sequential data. This section delves into the evolution of these mechanisms, tracing their journey from the inception in sequence-to-sequence models to the pivotal role in the groundbreaking Transformer architecture. By exploring various incarnations and applications of attention mechanisms, this section highlights the impact across diverse domains, such as natural language processing, computer vision, and healthcare. By dissecting these techniques and their adaptations, this thesis aims to provide a comprehensive understanding of their functionalities, limitations, and innovative solutions designed to overcome these challenges. Meanwhile, this section illustrates how these techniques have been practically applied within our experiment's context. It provides insights into the effectiveness of the attentional mechanisms discussed in terms of the practical application and the specific challenges in the research objectives.

3.1 Attention Mechanisms

3.1.1 Evolution of Attention Models

In 2014, Bahdanau et al. [42] introduced the attention mechanism to address the limitations of traditional sequence-to-sequence models, especially for long sequences. This mechanism allowed the model to dynamically focus on different parts of the input sequence when producing an output, leading to significant improvements in tasks such as machine translation. In other words, it allowed models to focus on specific parts of the input sequence when producing an output, similar to how humans pay attention to details when understanding a concept. Bahdanau et al. [42] presented an attention mechanism that fell under the category of soft (deterministic) attention. This mechanism constructed the context vector by taking a weighted average of all keys.

Following the pioneering work of Bahdanau et al. [42], there has been a swift evolution of attention model variants across diverse application domains. A notable feature of soft attention was its differentiability concerning the inputs. This characteristic ensured that the entire system could be optimized using conventional back-propagation techniques. In line with this, Xu et al. [43] introduced the concept of hard (stochastic) attention. In this approach, the context vector was derived from keys that were stochastically sampled. The hard attention model was more efficient than the soft attention model because it did not calculate attention weights for every element every time. However, this made it non-differentiable and harder to optimize. To train it, the research used methods such as an approximate variational lower bound or REINFORCE [44].

On this basis, Luong et al. [45] proposed global and local attention methods for machine translation tasks. The global attention closely resembled the soft attention mechanism. On the other hand, the local attention mechanism selectively considered only a portion of the source words at any given time. This made it more computationally efficient than global or soft attention. Furthermore, it maintained differentiability in most areas, which distinguished it from hard attention and facilitated its training and implementation.

Building on the foundation of the attention mechanism, Vaswani et al. [46] introduced the Transformer architecture in 2017. This model utilized a self-attention mechanism, allowing it to weigh input elements differently based on their relevance. The Transformer architecture has become the backbone for many state-of-the-art models in natural language processing. The Transformer architecture allowed the development of large-scale language models like Generative Pre-trained Transformer (GPT) by Radford et al. [47] and Bidirectional Encoder Representations from Transformers (BERT) by Devlin et al. [48]. These models have set new benchmarks in various NLP tasks—from text generation to sentiment analysis.

While attention mechanisms were initially popularized in NLP, their utility was soon recognized in computer vision. Researchers began focusing on convolutional neural networks, enhancing their ability to work on salient features in images. Minaee et al. [49] provided a comprehensive survey on using deep learning, including attention mechanisms, for image segmentation.

Forootan et al. [50] reviewed the application of machine learning and deep learning in energy systems, highlighting the potential of attention mechanisms in forecasting energy consumption and optimizing grid operations. Udendhran and Balamurugan [51] discussed the integration of deep learning architectures, including attention mechanisms, in smart farming applications, emphasizing their role in ensuring security and efficiency. In the realm of healthcare, Nguyen et al. [52] employed deep learning models with attention to forecasting the occurrence of dengue fever in Vietnam based on climate data, showcasing the potential of these models in predicting disease outbreaks.

3.1.2 Self-Attention

Having traced the development and widespread use of attention-based models, it becomes clear that these models significantly influence machine learning. However, understanding the evolution and impact is crucial. As moving forward, it is crucial to explore the technical intricacies and challenges posed by these models, especially when dealing with sequential data.

Recurrent models inherently restrict parallel processing within a single training instance due to their sequential nature [53]. This becomes particularly problematic for longer sequences, where memory limitations hinder the batching of multiple examples [54]. The introduction of attention mechanisms [55] has been a game-changer in sequence modeling, enabling the modeling of dependencies irrespective of the distance between sequences. However, it was noteworthy that, with a few exceptions [56], these attention

mechanisms were predominantly paired with a recurrent network.

The concept of self-attention, sometimes referred to as intra-attention, is a mechanism that interlinks different positions within a singular sequence to derive a representation of that sequence [56]. Parikh's work [55] introduced a self-attention mechanism specifically designed to align words in two sentences, emphasizing the most pertinent parts of each sentence for comparison purposes. In a contrasting approach, Vaswani [56] incorporated positional encodings before self-attention, enabling the model to harness the order of the sequence. This innovation has exhibited commendable performance, positioning it as a viable alternative to traditional recurrence or convolution techniques. Some studies have explored self-attention for time series forecasting. To address the challenges of indexing on the immutable blockchain system, Ma et al. [57] designed a hierarchical timestamp structure that supported efficient range queries on the timestamp field.

However, it is essential to know the potential pitfalls associated with the robust dynamics of positional self-attention. Given its comprehensive consideration of all sequence attributes, it can pose challenges, especially when dealing with time series data. Many researchers have focused on resolving those issues. Lee-Thorp et al. [58] introduced the FNet model, which leveraged Fourier Transforms as a replacement for the self-attention mechanism in Transformers, resulting in faster training time and comparable accuracy on various benchmarks. The approach emphasized the potential of using linear transformations to achieve efficient and effective model architectures.

In light of this, we promote the adoption of cycling encoding as a substitute for positional self-attention. This method refines the attention's focus based on the temporal characteristics of the target data, ensuring a more nuanced and contextually relevant representation.

3.1.3 Attention Model Implementation

The self-attention mechanism, employed in models such as Transformers, involves multiple attention heads within each self-attention sublayer. For input sequence, each attention head operates on an input sequence, $x = (x_1, \ldots, x_n)$ of n element where $x_i \in \mathbb{R}^{d_x}$, and computes a new sequence $z = (z_1, \ldots, z_n)$ of the same length where $z_i \in \mathbb{R}^{d_z}$. Each output element, z_i, is computed from the weighted sum of the input elements that have been linearly transformed:

$$z_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V) \quad [56]$$

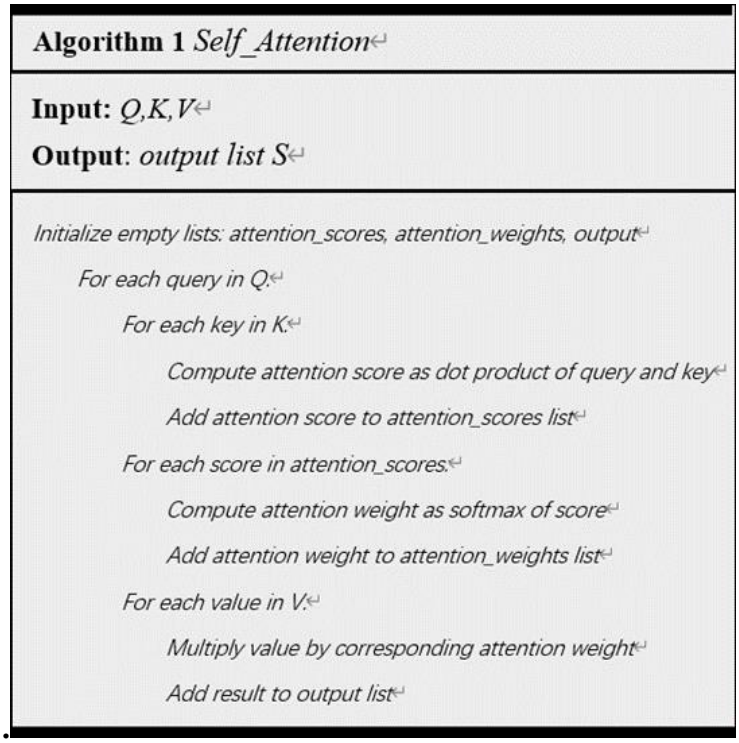
The weight coefficient α_{ij} for the contribution of input element x_j to output element z_i is computed using a SoftMax function:

$$\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^{n} \exp e_{ik}} \quad [56]$$

The SoftMax function is applied to a set of scores that measure the compatibility or similarity between x_i and x_j. Then scores are typically computed using a trainable linear transformation of the input elements. This involves dot products between query, key, and value vectors derived from the input elements. Finally, the output of each attention head (different z sequences) is concatenated and then linearly transformed through a

parameterized matrix to form the final output of the self-attention sublayer.

This self-attention mechanism allows the model to weigh the value of different parts of the input sequence differently for each output element, providing a flexible and powerful way to model relationships in the data.



Algorithm 1 *Self_Attention*

Input: *Q,K,V*

Output: *output list S*

```
Initialize empty lists: attention_scores, attention_weights, output
    For each query in Q:
        For each key in K:
            Compute attention score as dot product of query and key
            Add attention score to attention_scores list
        For each score in attention_scores:
            Compute attention weight as softmax of score
            Add attention weight to attention_weights list
        For each value in V:
            Multiply value by corresponding attention weight
            Add result to output list
```

Figure 1 Self-Attention Pseudocode

Figure 1 describes a function *Self_Attention* with three input matrices: *Q* (queries), *K* (keys), and *V* (values). It initializes three empty lists: *attention_scores, attention_weights*, and *output*. For each query in *Q*, it computes the dot product with each key in *K* to get the attention scores. These scores are then normalized using the SoftMax function to obtain the attention weights. Each value in *V* is then multiplied by the corresponding attention weight, and the results are added to the output list.

3.2 Auto-Encoder (AE)

Auto-encoders (AEs), as outlined in Goodfellow et al. [59], are a fundamental technique in unsupervised learning, utilizing neural networks to learn data representations. These tools are particularly adept at managing high-dimensional data, focusing on dimensionality reduction to effectively represent datasets. An auto-encoder comprises three core components: the encoder, the code, and the decoder. The encoder compressed the input into a code, which the decoder then used to reconstruct the input. This makes auto-encoders especially useful in learning generative data models, a concept further explored by Liu et al. [60].

Auto-encoders have extensive applications in unsupervised learning tasks, including dimensionality reduction, feature extraction, efficient coding, generative modeling, denoising, and anomaly detection, as discussed by Zhang et al. [61]. They are fundamentally similar to Principal Component Analysis (PCA), particularly in terms of reducing the dimensionality of large datasets, a similarity most apparent when comparing PCA to a single-layered auto-encoder with a linear activation function, as noted by Sarker et al. [62].

Regularized auto-encoders, such as sparse, denoising, and contractive types, are instrumental in learning representations for later classification tasks, as highlighted by Vincent et al. [63]. Variational auto-encoders (VAEs), on the other hand, are utilized as generative models, as detailed by Kingma and Welling [64]. These models map input data into the parameters of a probability distribution, making them effective for various

applications.

-Sparse Autoencoder (SAE) A sparse autoencoder, as described in Makhzani and Frey [65], incorporates a sparsity penalty on its coding layer, which is a crucial aspect of its training process. Unlike traditional autoencoders, Sparse Autoencoders (SAEs) can have a greater number of hidden units compared to their inputs. However, the key characteristic of SAEs is that only a limited number of these hidden units are allowed to be active simultaneously. This constraint develops a sparse model.

The structure of a sparse autoencoder in Figure 2 illustrates several active units in the hidden layer. Due to the sparsity constraint, the model is forced to adapt uniquely to the statistical features of the training data. This adaptation is essential for the model to effectively capture and represent the underlying patterns and characteristics of the data, despite the limitation on the number of active units in the hidden layer.

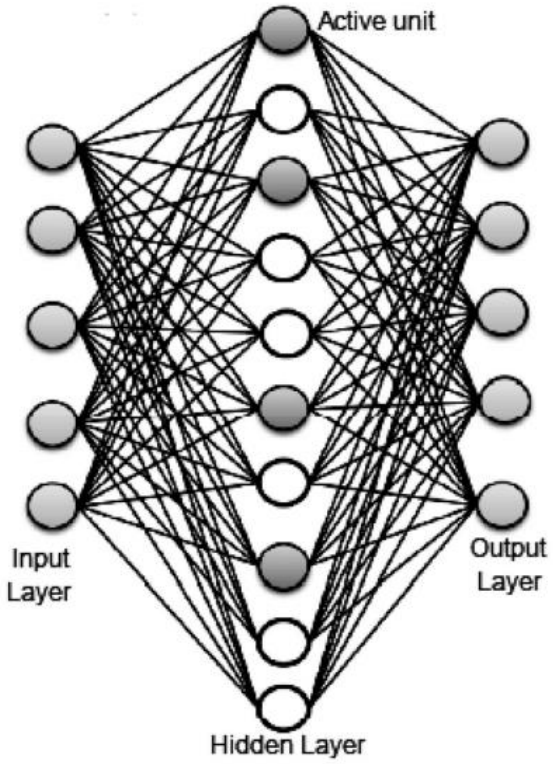


Figure 2 Schematic structure of a sparse autoencoder (SAE) with several active units (filled circle) in the hidden layer [65]

-Contractive Autoencoder (CAE) The Contractive Autoencoder, as proposed by Rifai et al. [66], is designed to enhance the robustness of autoencoders to small variations in the training dataset. This is achieved by incorporating a specific regularizer in its objective function. The regularizer's role is to ensure that the model learns an encoding that remains stable and consistent despite small changes in the input values. As a result, the sensitivity of the learned representation to the training input is significantly reduced.

-Variational Autoencoder (VAE) represents a unique and effective approach in the area of generative modeling, distinct from traditional autoencoders. Unlike conventional autoencoders that map input directly onto a latent vector, VAEs transform the input data into parameters of a probability distribution, such as the

mean and variance of a Gaussian distribution. This method is based on the assumption that the source data is governed by an underlying probability distribution, and the VAE aims to identify the parameters of this distribution [66].

VAEs have demonstrated versatility beyond unsupervised learning, finding applications in semi-supervised and supervised learning domains. The ability to model the underlying probability distribution of data makes them particularly effective for generative tasks. This approach, initially designed for unsupervised learning, has shown promising results in other areas, including semi-supervised learning and supervised learning scenarios [67-69].

3.3 Regression model

The term *regression* was first used by Galton [70] in the context of hereditary traits. Since then, the scope of regression analysis has significantly expanded, encompassing various types and techniques for more accurate and complex data modeling. Regression models have become a cornerstone of statistics used to understand relationships between variables and often to predict future observations. Therefore, our data preprocess will focus on regression tasks to evaluate their performance from our data. They include linear regression, polynormal regression, Support Vector Regression (SVR), and KNN regression.

-Linear Regression is one of the simplest and most widely used statistical techniques for predictive modeling and data analysis. It is a linear approach to modeling the

relationship between a dependent variable and one or more independent variables. Throughout the 19th and 20th centuries, linear regression evolved in parallel with advancements in computational power and statistical theory [71]. In particular, Galton's work [70] on regression to the mean in the context of genetics introduced the term regression in the late 19th century. Linear regression relies on several key assumptions including linearity, independence, homoscedasticity, and normality. The quality and validity of a linear regression analysis hinge on the satisfaction of these assumptions. Hoerl [72] introduced penalties on the size of coefficients to address issues of overfitting and multicollinearity, thereby enhancing the model's predictive performance. This approach, similar to Ridge Regression, applies regularization techniques to mitigate these problems. At its core, linear regression models the relationship between a dependent variable, independent variables X_i, and a random error term ϵ. It assumes that this relationship is linear and can be expressed as $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + .. + \beta_n X_n + \epsilon$, where β_0 is the intercept, β_i is the coefficient, and ϵ is the error term. Despite its simplicity, linear regression can provide accurate predictions, especially in scenarios where the linear assumption holds. It is also the foundation for understanding more complex models. One of the primary strengths of linear regression is its straightforward interpretability. The coefficients represent the average effect of a one-unit change in the predictor variable on the response variable, holding other predictors constant.

-Polynormal Regression is a regression analysis in which the relationship between

the independent variable x and the dependent variable y is modeled as an n^{th}-degree polynomial. It is used to understand more complex connections between variables than a basic straight-line relationship can display. Polynomial regression models are an extension of linear regression models. While the method of least squares had origins in the 18th century, the use of polynomial terms in regression was a natural progression as mathematicians and statisticians sought to fit more complex models to data [73]. They allow for a nonlinear relationship between the dependent and independent variables by introducing higher-order terms of the predictor variables in the model. The general form is:

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \cdots + \beta_n X^n + \epsilon \text{ [71]}$$

By including x-squared, x-cubed, etc., as additional predictors, polynomial regression allows for curvature in the modeled relationship, providing a more flexible fit to data that display non-linear trends [71]. Hastie et al. [74] stated a significant challenge with polynomial regression was overfitting, especially as the degree of the polynomial increases. Techniques such as cross-validation and regularization have been developed to mitigate this. Therefore, in choosing the right degree, various model selection criteria such as AIC, BIC, or cross-validation are used to select the appropriate degree of the polynomial to balance the bias-variance tradeoff. Polynomial regression's primary strength is its ability to model data with non-linear trends effectively. This is particularly useful in scientific data where relationships are rarely perfectly linear. Despite the non-linear fitting, polynomial models remain

relatively interpretable, especially for lower-order polynomials.

Polynomial regression extends the flexibility and applicability of linear regression by introducing the capability to model non-linear relationships. Its development and continued refinement have made it a valuable tool in the arsenal of statistical modeling techniques, allowing for more accurate and nuanced analysis of complex datasets.

- Support Vector Regression (SVR) is an extension of the Support Vector Machine (SVM), a widely recognized tool in machine learning and statistics for classification tasks. SVR utilizes the same principles as SVM but is applied to predict continuous outcomes, offering a robust and efficient prediction methodology. The concept of Support Vector Machines was introduced by Vapnik et al. [75] in the 1960s. SVR was later developed as an extension of the classification model to handle regression tasks. SVR works via mapping input features into high-dimensional feature spaces and then finding a linear model that fits the data. Unlike traditional regression models that minimize the residual sum of squares, SVR focuses on fitting the error within a certain threshold, seeking to minimize the generalization error rather than the training error. A key concept in SVR is the introduction of a margin of tolerance ϵ around the prediction line. The algorithm then attempts to fit the model within this margin while also minimizing the model complexity [76]. Over the years, SVR has been further refined, particularly in how it handles non-linear relationships through kernel functions, allowing it to model complex datasets effectively. A significant

advancement in SVR was the introduction of the kernel trick, allowing the algorithm to fit non-linear models by implicitly mapping input features into high-dimensional feature spaces. Because of using kernel functions, SVR can model non-linear relationships as effectively as linear ones, providing a flexible approach adaptable to various types of data. SVR is known for its robustness in predictive modeling, especially in situations where the dataset has a lot of noise or the number of dimensions was greater than the number of samples. Drucker et al. [77] indicated SVR often provides superior predictive performance, especially in cases where the relationship between the dependent and independent variables was complex and non-linear.

SVR represents a powerful and flexible approach to regression analysis capable of delivering high-quality predictive performance, especially in complex and high-dimensional datasets. Its development from the principles of SVM and continuous enhancements have established it as a key technique in the repertoire of statistical learning methods.

- K-Nearest Neighbors Regression (KNN Regression) is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until function evaluation. It is a non-parametric method used for regression and classification, relying on the proximity of sample points to make predictions [78]. KNN Regression predicts the value of a new point based on the 'k' nearest neighbors. The prediction is typically the average of the values of its nearest

neighbors. It involves the selection of a distance metric (like Euclidean or Manhattan distance) to identify the closest neighbors. The choice of ‘k’, the number of neighbors (usually 5), is crucial in determining the effectiveness of the KNN model [79]. Over the years, KNN has evolved with the introduction of various weighting schemes, distance metrics, and algorithms to improve its efficiency and accuracy, especially in large and complex datasets. Innovations such as KD trees and Ball trees have been developed to improve the computational efficiency of KNN especially important for large datasets. KNN Regression is popular for its simplicity and has been widely used in situations where the data is rich in structure but difficult to model parametrically. Meanwhile, KNN is fairly intuitive and interpretable as it makes predictions based on observable and similar instances [80]. Its non-parametric nature makes it versatile for various kinds of data, especially when the relationship between variables is unknown or complex.

K-Nearest Neighbors Regression remains an important tool in statistical learning, known for its simplicity and effectiveness in multiple scenarios. Despite its computational intensity with larger datasets, continued methodological enhancements and applications in various fields underscore its enduring relevance and adaptability.

4. Proposed Methodology

In the previous chapters, we focused on the multifaceted nature of time series data and discussed its significance, the complexities it embodied, and the limitations inherent in conventional preprocessing methods. Despite the advancements in neural network architectures and their applications to temporal data, a significant gap remains in accurately and efficiently capturing the intricate temporal dependencies, particularly in periodic data. The challenge is not just to process the data but to do so in a manner that is time-efficient, adaptable, and capable of harnessing the full potential of the temporal information embedded within. Addressing these challenges requires a novel approach that goes beyond the traditional paradigms of time series data preprocessing.

This chapter introduces a pioneering methodology to enhance the preprocessing of time series data. This innovative approach aims to integrate an autoencoder with a cycling layer self-attention mechanism, a synergy crafted to capture, process, and utilize temporal patterns in a manner previously unattained. More specifically, the methodology intends to significantly improve the accuracy of predictions and insights from the time series data and streamline the processing efficiency, thereby enabling more rapid and adaptable data analysis.

The proposed methodology is a testament to the fusion of autoencoding techniques and

attention mechanisms to tackle the unique challenges of time series data. The autoencoder, known for its proficiency in data dimensionality reduction and feature learning, is tailored to suit the needs of time series analysis. Meanwhile, the cycling layer self-attention mechanism introduces a novel approach to capturing temporal dependencies, allowing the model to focus on relevant segments of data cyclically and iteratively, enhancing the model's understanding and representation of time-bound patterns.

In the forthcoming section, we will dissect the intricate architecture of the proposed methodology, explore its components, and articulate the rationale behind its design.

4.1 Positional Layer Self-attention

Incorporating Positional Layer Self-attention into data preprocessing primarily targets leveraging its ability to understand and encode the positional relationships within sequential data effectively. Traditional methods of sequence data processing often rely on manual feature engineering or assuming a simplistic approach to a sequence, which can lead to a significant loss of context and information. Contrarily, Positional Layer Self-attention automates the recognition of these complex dependencies, which could avoid the disadvantages of the traditional approaches.

Data preprocessing enhances the performance of self-attention mechanisms by ensuring that the temporal or sequential relationships intrinsic to the data are not merely preserved but are accentuated and actively utilized for analysis or predictions. This

automated, nuanced understanding of sequence directly translates to improved model performance, particularly in tasks where the order of data points is critical. The ability of models to discern subtleties in sequence aids in more accurate predictions, richer data representations, and, ultimately, more valuable analyses.

The position of data points in a sequence can dramatically affect the Interpretation and outcome of the analysis. In the prediction of weather patterns from our data, the positions of each data point relative to others encapsulate crucial temporal dependencies and patterns. Therefore, recognizing positional dynamics allows models to capture the contextual flow and dependencies that exist between data points, which is pivotal for accurate analysis. However, traditional manual preprocessing methods often fail to adequately capture and maintain the positional integrity of data because traditional manual preprocessing methods often inadequately preserve the positional integrity of data, leading to loss of context, potential data distortion, scalability, and flexibility issues which will ultimately lead to analysis inefficiency.

Traditional self-attention mechanisms enable models to weigh the essence of the different parts of the input data for each output element. However, without a sense of order or position, the model treats the sequence as if all parts are equally positioned, losing out on the temporal or sequential context that is often critical in interpretation and analysis. Positional Layer Self-attention addresses this by incorporating positional information. Firstly, positional encoding is used to inject positional information into the model. Our approach involves using sine and cosine functions of different frequencies

to encode each position. These functions provide a continuous and differentiable method to represent the position, which is crucial for model training. Each position generates a unique positional encoding vector, which is then added to the input embedded with the corresponding sequence. This process ensures that the model understands the content of each element and its position within the sequence.

Algorithm 2 *Positional_Encoding*

Input: *sequence_length, dimensions*
Output: *pos_enc*

```
Initialize pos_enc as a matrix of size sequence_length x dimensions
with all zeros

    For pos from 0 to sequence_length - 1:
        For i from 0 to dimensions - 1 in steps of 2:
        pos_enc[pos, i] = sin(pos / (10000^(2*i/dimensions)))
        If i+1 < dimensions:
            pos_enc[pos, i+1] = cos(pos /
        (10000^(2*i/dimensions)))
```

Figure 3 Positional_Encoding Pseudocode

Figure 3 describes a function *Positional_Encoding* that takes the length of the sequence (*sequence_length*) and the number of dimensions (dimensions) as input. It initializes a matrix *pos_enc* of size *sequence_length* x dimensions with all zeros. Then, it iterates over each position in the sequence and each dimension in the embedding space. For each pair of positions and dimensions, it computes a value using the sine or cosine function and assigns those values to the corresponding element in the *pos_enc* matrix. The function finally returns the *pos_enc* matrix, which represents the positional

encoding of the sequence.

The algorithm starts by initializing a matrix *pos_enc* of size *sequence_length x* dimensions with all zeros. This operation is *O (sequence_length * dimensions)* because it involves setting up a matrix with *sequence_length * dimensions* elements. The outer loop runs from *0* to *sequence_length – 1*, iterating over each position in the sequence. This is *O(sequence_length).* The inner loop runs from *0* to *dimensions – 1* in Step 2. Essentially, this loop runs *dimensions/2* times because it increases the counter *I* by 2 in each iteration. This is *O(dimensions).* Given these steps, the overall time complexity of the algorithm is the product of the complexities of the nested loops, since they represent the most significant amount of work done by the algorithm. Thus, the time complexity is *O (sequence_length * (dimensions/2)),* which simplifies to *O (sequence_length * dimensions).*

The Integration of positional encoding of the self-attention mechanism is a pivotal aspect of Positional Layer Self-attention. Once the positional encodings are added to the input embeddings, the modified embeddings are fed into the self-attention layer. The self-attention mechanism then computes the attention scores, considering the inherent features of the elements and their positional encodings.

This thesis utilizes Positional Layer Self-attention to capture temporal dependencies within data more effectively. Its contribution lies in enhancing the model's ability to discern intricate patterns over time, thus improving the accuracy of predictions. This self-attention mechanism offers a more refined approach to analyzing time-series data,

providing clearer insights and a robust foundation for subsequent analytical processes.

4.2 Cycling Layer Self-attention

The advent of Cycling Layer Self-attention represents a new shift in unsupervised machine learning, particularly in the preprocessing of time series data. The motivation for this novel architectural layer stems from the intrinsic need to capture the cyclical nature of temporal data more effectively. Traditional layers, such as positional encodings, are used to handle sequential information but often lack the finesse to grasp the cyclical patterns paramount in datasets, such as weather patterns, where features fluctuate periodically.

The Cycling Layer is designed to fill this gap. It is different from the one-size-fits-all approach from the traditional positional encoding but provides a framework to isolate and understand the seasonal and daily cycles inherent in time series datasets. The flexibility of the Cycling Layer allows manual adjustment of periodic time features, enabling a more tailored and precise capture of cyclical trends. This capacity to adjust and focus on specific cycle lengths with less risk of overfitting to irrelevant patterns is an advancement over traditional methods. Furthermore, the simpler construction of the Cycling Layer compared to more complex positional encoding mechanisms can potentially reduce computational overhead and streamline the preprocessing phase, leading to cost efficiencies in time and resources.

Algorithm 3 *Cycling_Encoding*

Input: *batch_size, sequence_length, embed_dim*
Output: *combine feature x*

```
x is assumed to be a tensor of shape (batch_size, sequence_length,
embed_dim)
where each value in x is a timestamp. The goal is to encode these
timestamp
    For pos from 0 to sequence_length - 1.
        Convert x to fraction of the day (x / (cycling period time))
modulo 1
            Convert to radians x * 2 * pi
            Concatenate([sin(x), cos(x)], along last dimension)
            Concatenate original features with cyclically encoded
features
```

Figure 4 Cycling_Encoding Pseudocode

The Cycling Layer is a novel construct in the architecture of machine learning models designed for processing time series data. Figure 4 begins with the input tensor *x*, which is assumed to have a shape of (*batch_size, sequence_length, embed_dim*), where each value in x represents a timestamp. The design goal of the Cycling Layer is to encode these timestamps to make the cyclical information explicit and usable by the model.

The cycling position features are the cornerstone of the Cycling Layer's functionality. They are derived from the radian-converted timestamps using sine and cosine functions, creating a pair of values for each point in time. This duality allows the model to capture the cyclical nature of time, ensuring that it recognizes the continuity at the cycle's boundaries—such as the transition from the end of one day to the beginning of the next. The sine and cosine values are concatenated along the last dimension of the tensor, effectively doubling the size of this dimension. This concatenation preserves the

original timestamp information while augmenting it with cyclical encoding, providing a dual representation of linear time and its inherent cycles.

The final step in the Cycling Layer's process is to concatenate the original features with these newly created cyclical position features. The result is a combined feature vector that retains all information from the original data enriched with a clear, cyclical context. This enriched representation is then fed into the self-attention mechanism, which can now take advantage of the cyclical patterns alongside the raw data features.

Considering all the operations (sine and cosine functions and concatenation) are done for each element in the batch, and there are *batch_size* such sequences (should be ignored), the overall time complexity of the algorithm when considering the batch size is *O(sequence_length * 2* dimensions).* Since *embed_dim* is a property of the model architecture and is typically a constant, and *2* is a constant factor, the time complexity simplifies to *O (sequence_length * dimensions)*, which is the same as the positional layer.

Overall, Cycling Layer Self-attention, as presented in this thesis, distinguishes itself from traditional methods by its iterative refinement process, offering more precise feature extraction that considers time trends. This method delves deeper into data intricacies, yielding a richer, more accurate representation. Its contribution is evident as it can analyze the data more comprehensively, setting a new standard for precision in understanding complex patterns, which is a significant stride beyond the capabilities of conventional methodologies.

4.3 Autoencoder Preprocessing

In our experiment, the cycling layer will be implemented in an autoencoder, a type of artificial neural network used for learning efficient coding of input data. The autoencoder will be designed to encode input data into a compressed representation, and then decode this representation into the original format. The cycling layer and self-attention layer will be integrated into the encoding process, allowing it to capture and encode the cyclical patterns in the time series data [80].

In data preprocessing, particularly for time series data, autoencoders serve a dual purpose. Firstly, they act as powerful denoising agents. Time series datasets often come with a significant amount of noise, which can obscure the underlying patterns – critical for accurate analysis and prediction. An autoencoder can learn to filter out this noise and retain the critical signals, thereby purifying the data for better analytical outcomes. Secondly, autoencoders are instrumental in dimensionality reduction. Time series data can be incredibly high-dimensional, especially when recorded over long periods or at high frequencies. Processing such vast amounts of data can be computationally expensive and time-consuming. Autoencoders help in distilling the data into a more manageable form without losing the essential temporal dynamics. This reduced representation improves computational efficiency and model performance by mitigating the risk of overfitting, which is common in high-dimensional data scenarios. The utility of autoencoders in preprocessing is further magnified when dealing with multivariate time series data, where the interactions between different variables can be

intricate and non-linear. By learning to encode these interactions, autoencoders can unveil the underlying structure of the data, providing a cleaner, more structured input for subsequent models and analyses.

Within the encoding phase of the autoencoder, we have innovatively integrated the cycling layer and self-attention layer. This integration is critical as it not only allows the encoder to reduce the data dimensionality but also to recognize and encode the cyclical patterns inherent in time series datasets.

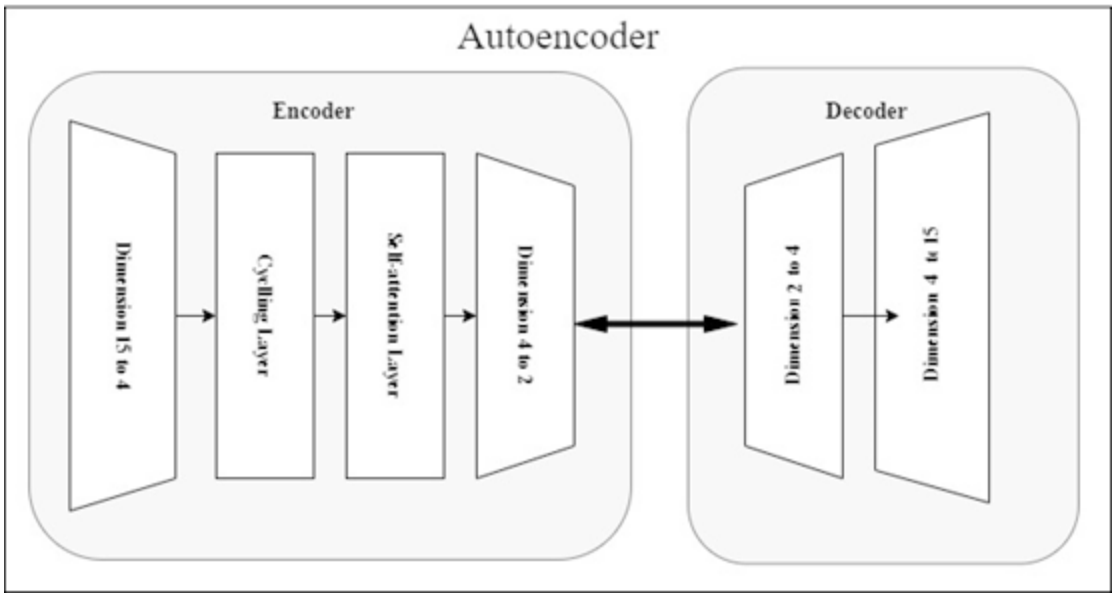


Figure 5 Autoencoder with cycling layer

In Figure 5, we can see encoding process begins with the input of raw data, which potentially consists of various time series variables recorded over time. The data is first passed through the cycling layer, which embeds cyclical temporal information into the data representation. The enhanced data then enters the self-attention layer, which allows the network to prioritize information based on the cyclical context provided by the

previous layer. As the data progresses through subsequent layers of the encoder, it is compressed into a latent space—the heart of the autoencoder. This space represents the most significant features of the input data in a compacted form. It is at this bottleneck where the data, now a compressed representation of the original data set, holds the most critical patterns and trends that are learned by the network.

The decoding process is the mirror image of encoding. The compressed representation from the latent space is taken as input by the decoder. The decoder works to reconstruct the original high-dimensional data from this compressed form. The reconstruction is done by passing the encoded representation through a series of layers that progressively increase in size.

The initial layers of the decoder aim to re-expand the encoded features back into the format closer to the original input data, while subsequent layers refine this reconstruction, adjusting the output based on the loss function that measures the difference between the original data and the reconstruction. The final output layer of the decoder is designed to match the dimensions of the original input data, where the reconstructed data emerges, ready to be compared against the original dataset to assess the fidelity of the reconstruction and the efficiency of the encoding process.

In our experiment, the entire process of the autoencoder, from encoding through decoding, is subject to optimization and training. The loss function, such as mean squared error for continuous data, is often used as a measure to guide the adjustment of network weights to minimize reconstruction error, ensuring that the autoencoder learns

the most effective representation of the input data. Through iterative training, the autoencoder refines its parameters until the reconstruction loss is minimized, signifying that the network has learned a robust representation of the time series data.

The autoencoder preprocessing we proposed is adept at isolating core features from complex datasets, enabling models to focus on the most impactful elements. This selective focus leads to enhanced model accuracy and interpretability, setting this method apart from traditional preprocessing techniques by providing a clearer, more refined data perspective for in-depth analysis.

4.4 Data Split

The strategic partitioning of the dataset into training, validation, and test sets is pivotal in developing a predictive model. The training set enables the model to learn and internalize data patterns, comprising the bulk of the data for comprehensive learning. The test set, separated from the model's training phase, serves as an unbiased metric for evaluating the model's ability to generalize to new data. Validation subsets extracted from the training data act as checkpoints for model refinement, providing a safeguard against overfitting and ensuring the model's performance is robust and reliable.

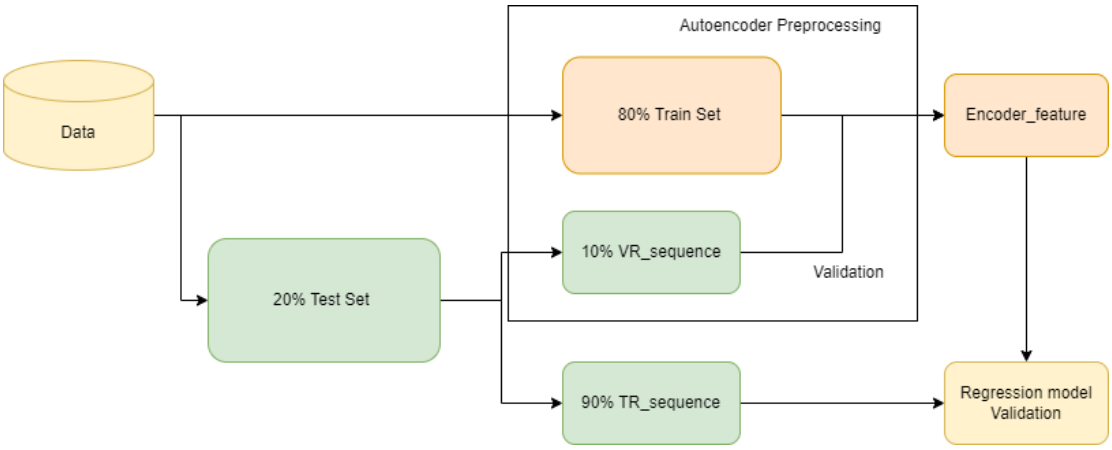


Figure 6 Data Split Flow Graph

As indicated in Figure 6, the data splitting strategy commences with the segregation of the entire dataset into an 80-20 split. Eighty percent is allocated to the training set, forming the basis of the autoencoder preprocessing. This subset undergoes a transformative process within the autoencoder, learning to encode the input data into a compressed representation, which is instrumental for feature extraction.

The remaining 20% of the dataset is earmarked as the test set, acting as the final valuator of the model's performance. Further refinement within the test set splits it into Validation (*VR_sequence*) and Test (*TR_sequence*) partitions.

The *VR_sequence*, a tenth of the training set, is looped back into the autoencoder as a validation set, allowing for model validation and early stopping—a technique employed to prevent overlearning and overfitting. As the autoencoder iteratively learns to compress and reconstruct the training data, the *VR_sequence* serves as an unbiased indicator of how well the encoded features generalize beyond the training examples. It is a vital tool in calibrating the model, guiding the hyperparameter tuning, and determining the ideal complexity of the network's architecture. Meanwhile, the early

stopping mechanism operates by continuously monitoring the validation loss, which is the error metric on the *VR_sequence*. This whole process serves as the iterations of training progress. If the validation loss fails to decrease or improves only marginally over a specified number of iterations, which is known as the 'patience' parameter, the training is halted. The model's parameters are then reverted to a state where the validation loss is at its minimum, ensuring that the model retains the best-learned features without being tainted by overfitting. Furthermore, in our autoencoder setup, learning rate adjustments are made in conjunction with early stopping. If the validation loss does not improve over time, the learning rate is reduced to refine the model's search for the optimal weights. This technique, often referred to as learning rate annealing or decay, allows the model to take smaller steps when approaching the loss function's minimum, leading to more precise convergence.

The *TR_sequence*, constituting the remainder of the test set, is utilized for final validation against various regression models. It provides a secondary checkpoint in the post-training phase to ensure that the encoding layer's learned representations are not overfitted and remain accurate with the data excluded from the initial validation during the training. In this phase, the encoded features crafted and refined through the autoencoder's processing are tested against various regression models. The models selected various regression models, such as linear regression, polynomial regression, Support Vector Regression, and K Nearest Neighbors regression, represent a spectrum of approaches that vary from the simple and interpretable linear models to the complex

and non-linear SVR and KNN. Therefore, the *TR_sequence* serves as the dataset in which these models are trained and evaluated, providing a comprehensive view of how well the encoded features can predict outcomes across different modeling techniques. For example, it represents how well the encoded feature can predict outcomes across different modeling techniques. The performance of these models on the *TR_sequence* indicates the quality of the encoded features that can capture the essence of the original data and enable these regression models to make accurate predictions.

Figure 6 illustrates this meticulous approach, outlining the process from the autoencoder preprocessing phase to the regression model validation phase. This systematic partitioning ensures the encoded features extracted from the autoencoder are robust, representative, and reliable for subsequent regression modeling.

5. Results and Discussion

5.1 Data Sample Overview

The experiment data used in this thesis was from Kaggle competition support. The Jena Climate dataset, from Kaggle, represents a comprehensive weather dataset recorded at the Weather Station of the Max Planck Institute for Biogeochemistry in Jena, Germany. The dataset was primarily collected for climate and environmental research, offering a detailed, long-term record of various meteorological conditions. The dataset for analysis spans over eight years from January 1st, 2009, to December 31st, 2016. The selection of this specific period allows observation and analysis of the impacts of different seasons, year-over-year variations, and significant weather events over eight years. Furthermore, taking advantage of data across multiple years is beneficial for understanding the cyclical pattern of the weather. Additionally, it helps make more accurate predictions about future weather conditions based on historical patterns. Furthermore, Jena's position is at the Saale Valley. This unique position influences its meteorological conditions, such as temperature variations, precipitation levels, and wind patterns. By focusing on this particular geographic location, the dataset offers a detailed and nuanced view of the local climate and provides insights relevant to the

area’s ecological, agricultural, and urban development specifically.

Feature	Minimum	Maximum	Mean
p (mbar)	913.60	1015.35	989.21
T (degC)	-23.01	37.28	9.45
Tpot (K)	250.60	311.34	283.49
Tdew (degC)	-25.01	23.11	4.96
rh (%)	12.95	100.00	76.01
VPmax (mbar)	0.95	63.77	13.58
VPact (mbar)	0.79	28.32	9.53
VPdef (mbar)	0.00	46.01	4.04
sh (g/kg)	0.50	18.13	6.02
H2OC (mmol/mol)	0.80	28.82	9.64
rho (g/m³)	1059.45	1393.54	1216.06
wv (m/s)	-9999.00	28.49	1.70
max. wv (m/s)	-9999.00	23.50	3.06
wd (deg)	0.00	360.00	174.74

Table 1 Range of each data features

Table 1 displays the overall range and mean for each feature reflecting distinct aspects of the atmospheric state. Atmospheric pressure indicates the weight of air, varying from *913.60 to 1,015.35* mbar, which is crucial in shaping weather patterns. Temperature ranges from *-23.01°C to 37.28°C*. The temperature range provides information on atmospheric stability, and dew point temperature, which is a key indicator of atmospheric moisture. Relative humidity, with its span from *12.95% to 100%*, impacts weather conditions and ecological systems. Water vapor metrics, including maximum water vapor pressure, actual vapor pressure, vapor pressure deficit, specific humidity, and water vapor concentration, collectively delineate the moisture level of the air,

influencing precipitation, cloud formation, and overall atmospheric processes. Air density, affected by temperature, pressure, and humidity, has implications across aviation to engineering. Wind speed and direction, despite some data anomalies, are fundamental in understanding air movement, weather systems, and pollutant dispersion. The range and mean of these variables illustrate the dynamic and complex nature of the climate at the Jena station, offering deep insights into weather patterns and providing a robust foundation for extensive climatic research and applications in forecasting, environmental planning, and beyond.

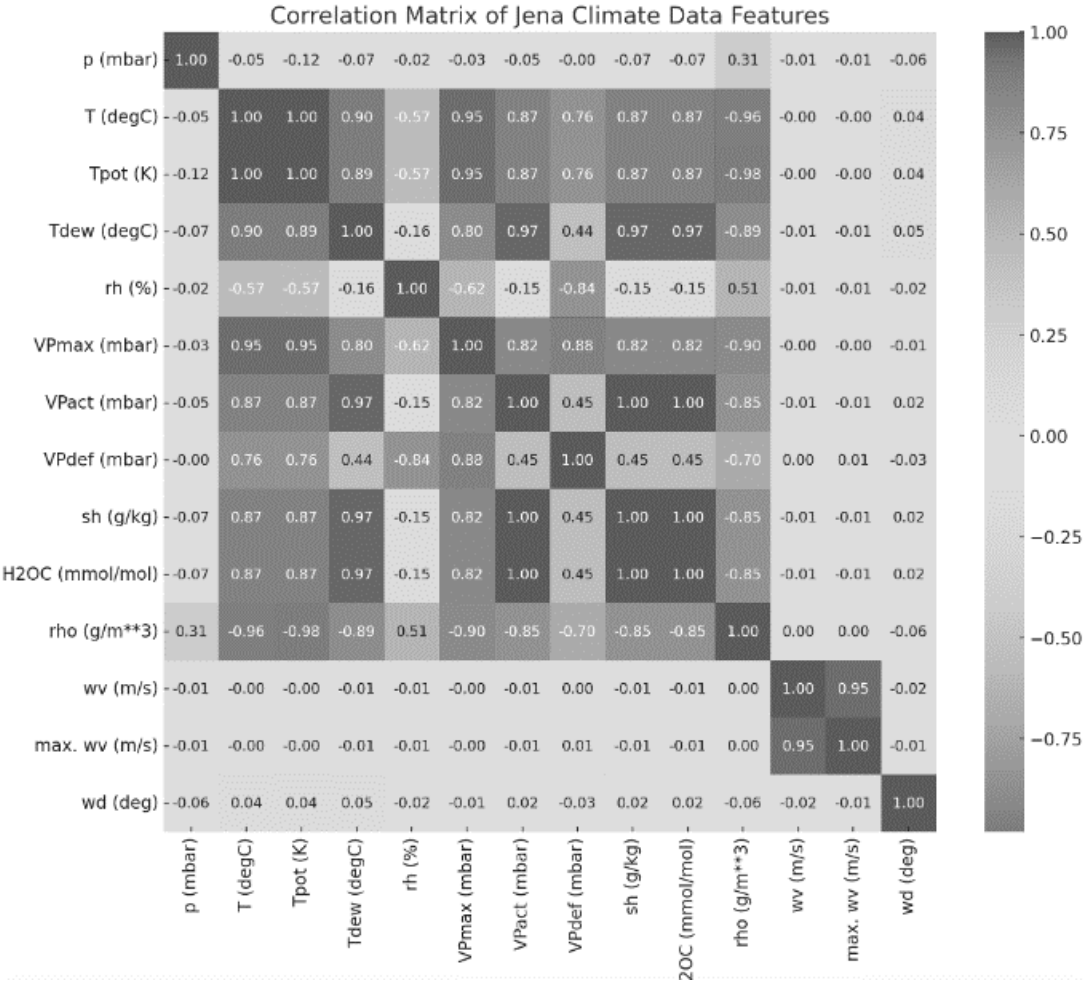


Figure 7 Correlation matrix of the Jena Climate dataset

On the other hand, to review the characteristics of each feature relationship, Figure 7

specifies that temperature (*T_degC*) demonstrates a strong positive correlation with potential temperature (*Tpot_K*) and dew point temperature (*Tdew_degC*), indicating a cohesive relationship among different measures of atmospheric heat. The specific humidity (*sh_g/kg*), water vapor concentration (*H2OC_ mmol/mol*), and actual vapor pressure (*Vpact_mbar*) are closely interrelated, reflecting the intertwined nature of moisture variables. Conversely, relative humidity (*rh_%*) inversely correlates with temperature, embodying the dynamic balance between temperature and moisture-carrying capacity of air. Atmospheric pressure (*p_mbar*) negatively correlates with temperature to some extent, in line with the thermal expansion of air. Wind speed (*wv _m/s*) and maximum wind speed (*max. wv_m/s*), while showing low correlation with other variables, emphasize that the nature of wind is likely independent of other meteorological factors. However, data anomalies should be noted for accurate interpretation. The dew point temperature (*Tdew_degC)* aligns closely with air temperature, reinforcing the relationship between temperature and moisture content. Lastly, the vapor pressure deficit (*Vpdef_mbar*) stresses its role in understanding air's drying capacity, positively correlating with temperature and negatively correlating with relative humidity, essential for evaporation and plant transpiration studies. Together, these relationships underscore the intricate and interdependent dynamics of climatic factors within the dataset.

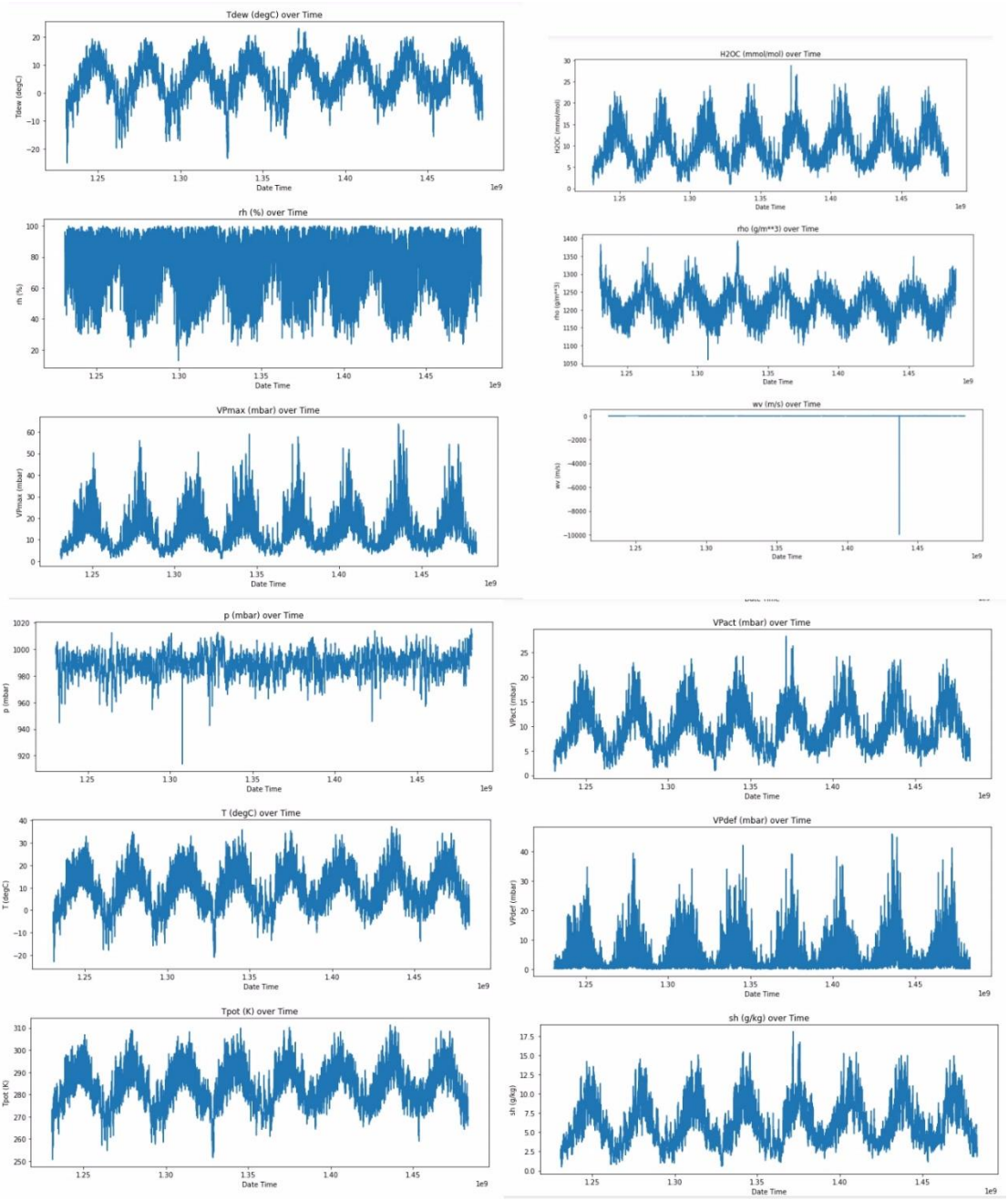


Figure 8 Visual Plot of Each Feature over Time

The series of visual plots provided in Figure 8 reflect key characteristics of time series data, as captured in the Jena Climate dataset. Each plot represents a different meteorological variable, and they exhibit clear time series features. Many plots demonstrate patterns that repeat regularly. This is indicative of seasonality—a common

feature in climate data where certain patterns (such as temperature increases or decreases) recur consistently each year. For example, the temperature plot shows cyclical behavior consistent with the expected seasonal weather changes. Meanwhile, the plots reveal varying degrees of volatility in different variables. For instance, wind speed and vapor pressure deficit display sharp spikes at certain points, those as indicators of sudden weather events or sensor anomalies. Furthermore, particularly in the wind speed plot, there are extreme values that deviate significantly from the rest of the data. These could be outliers due to extraordinary weather events or errors in data recording.

In conclusion, the Jena Climate dataset represents a highly complex and multidimensional data structure, characterized by its diverse variables. Each variable embodies different trends, ranges, and underlying relationships. This intricately woven tapestry of data reveals the dynamic interplay between various meteorological elements. Each feature, such as temperature and wind speed, interacts subtly yet significantly with others, shaping the overall climatic narrative. The temporal aspect of the data, with its detailed time series spanning over decades, adds a layer of depth, allowing the observation of long-term patterns, seasonal shifts, and transient anomalies. The presence of anomalies and outliers, such as the unrealistic wind speed values, allows for advanced models such as autoencoders to learn a normative pattern of the data, and thereby identify and isolate these anomalies. The rich temporal and meteorological diversity of the Jena Climate dataset, combined with its complexity and volume, makes

it an ideal candidate for unsupervised learning techniques like autoencoders with self-attention. These models can uncover deep, nuanced insights and patterns within the data, leading to improved analysis, more accurate predictions, and a better understanding of climate dynamics.

5.2 Evaluation Criteria

The evaluation of autoencoder models uses a comprehensive approach. The following section explains two distinct aspects.

5.2.1 Model Fitting Speed Comparison

Comparing the model fitting speed allows a comparison of the efficiency of autoencoder models employing traditional positional layer self-attention against those using cycling layer self-attention. This comparison is vital as it reveals the practicality and performance of the models in real-world applications, particularly in handling large and complex datasets like the Jena Climate dataset. Understanding which configuration yields faster convergence to the best fit can significantly impact the choice of model architecture in future implementations and research.

The key metric for this comparison Is the overall number of iterations taken by each model to reach the best fit, which is determined by the lowest training loss achieved. This metric is chosen as it directly reflects the time efficiency of the model in learning from the data and reaching a point of optimal performance.

5.2.2 Regression Model Performance Comparison

This analysis aims to evaluate and compare the performance of different regression models that utilize the representations learned by autoencoder models. This evaluation is crucial in understanding how effectively the learned representations can predict or reconstruct relevant outputs, thereby determining the practical utility of the autoencoder models in real-world applications. This comparison uses MSE, MAE, RMSE, and median MAE as the metrics for different regression methodologies.

-***Mean Squared Error (MSE)*** is the average of the squares of the errors or deviations, that is, the difference between the estimator and what is estimated. It is a measure of the quality of an estimator—it is always non-negative, and values closer to zero are better. In the context of climate data, a lower MSE indicates the better ability of the model to forecast weather variables. The algorithm is:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$ [81]

where n is the number of observations, Y_i is the actual value of the observation, and $\hat{Y}_i$ is the predicted value.

-*Mean Absolute Error (MAE)* measures the average magnitude of the errors in a set of predictions, without considering their direction. It's a linear score, which means all individual differences are weighted equally in the average. For climate predictions, a lower MAE signifies a model's effectiveness in capturing different weather phenomena without significant errors. The algorithm is:

$$MAE = n1\sum i = 1n \mid Yi - Y\hat{}i \mid$$ [81]

where n is the number of observations, Y_i is the actual value of the observation, and $\hat{Y}_i$ is the predicted value.

-Root Mean Squared Error (RMSE) is the square root of the mean square error. It is often preferred over MSE in some contexts because it is interpretable in the same units as the response variable. It gives a relatively high weight to large errors, reflecting the square quality of MSE. In predictive weather models, lower RMSE values indicate a closer fit to the data. The algorithm is:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}$$ [81]

where n is the number of observations, Y_i is the actual value of the observation, and $\hat{Y}_i$ is the predicted value.

-Median MAE While MAE provides an average measure of absolute errors, Median MAE focuses on the median of these errors, thereby reducing the influence of outliers significantly. In scenarios where predictions might be skewed by anomalies or extreme events, Median MAE offers a more resilient measure of central tendency. The algorithm is:

$$Median\ MAE = median(|\ Y_1 - \widehat{Y_1}\ |, |\ Y_2 - \widehat{Y_2}\ |, \ldots, |\ Y_n - \widehat{Y_n}\ |)$$ [81]

where n is the number of observations, Y_i is the actual value of the observation, and $\hat{Y}_i$ is the predicted value.

Each of these metrics provides different insights into the accuracy and performance of regression models, allowing analysts to understand various aspects of model prediction quality. They are fundamental in machine learning for model evaluation, comparison,

and selection.

5.3 Transformation Preprocessing

This section elucidates the preprocessing steps undertaken to prepare the Jena Climate dataset for subsequent analysis using an autoencoder model. This section details the conversion of the date-time column from its original form into a Unix timestamp, thereby standardizing it for easier model ingestion and alignment with other time series data processing methods. Also, we aim to address the problem of missing values, a common issue in real-world datasets. If this problem is left unaddressed, it can significantly distort model training and predictions. The chosen method, mean imputation, replaces missing values with the mean of each column, maintaining the overall statistical characteristics of the dataset. Lastly, this section discusses the creation of sequences from the processed data. Autoencoders, especially those designed for time series data, benefit from learning sequential patterns and dependencies, necessitating the transformation of the dataset into a sequence of data points.

5.3.1 Convert Date-Time Column

The process of converting the date-time column begins with transforming the original string representations of date and time into Python's datetime objects using the *pd.to_datetime* function. This function is versatile and can handle various string formats by specifying the exact format of the date and time in the dataset. For the Jena Climate dataset, the format is designated as *'%d.%m.%Y %H:%M:%S'*, which corresponds to

the Day, Month, Year, Hour, Minute, and Second structure of the date-time entries.

$$Unix\ epoch = \frac{DataTime - Timedelta}{Timedelta}$$

* $Timedelta\ start\ from$ "1970 − 01 − 01"

After converting to datetime objects, these objects are transformed into Unix timestamps. Unix timestamp is a commonly used standard for representing a point in time as the number of seconds that have elapsed since the Unix epoch *(00:00:00 UTC on 1 January 1970)*. This conversion is achieved by subtracting a standard epoch (in this case, the Unix epoch) from the datetime object and then dividing by the time delta to get the total seconds. The resulting Unix timestamps are numerical and thus more amenable to various computational and modeling techniques used in machine learning.

	Date Time
0	01.01.2009 00:10:00
1	01.01.2009 00:20:00
2	01.01.2009 00:30:00
3	01.01.2009 00:40:00
4	01.01.2009 00:50:00

	Date Time
0	1230768600
1	1230769200
2	1230769800
3	1230770400
4	1230771000

Figure 9 Before and After Conversion

Figure 9 illustrates the data before and after the conversion. Initially, the date-time column contains human-readable strings representing each data point's date and time. After conversion, the same column reflects a series of integer values, each representing the Unix timestamp corresponding to the original date and time. A table or snapshot with a few rows of data pre- and post-conversion would effectively demonstrate the change, providing a clear visual representation of how the raw temporal data is transformed into a structured, model-ready format. For example, the before data time

No. 0 will transform from *01.01.2009 00:10:00* string to *1230768600* number. This transformation is critical for standardizing the temporal data, ensuring it is in a consistent, numerical format optimal for computational processing in subsequent modeling stages. The figure effectively showcases this vital preprocessing step, emphasizing its significance in enhancing data structure and model efficiency. Furthermore, as opposed to no preprocessing, preprocessing significantly enhances data uniformity and model compatibility. It simplifies calculations and algorithms' interpretation of time, ensuring a consistent numerical format, which not only boosts computational efficiency but also increases the model's predictive accuracy, as it operates on standardized and error-minimized data, a clear advantage over using non-standardized, raw date-time formats.

5.3.2 Handling Missing Values

We choose the mean imputation to handle the missing values in the dataset. Mean imputation is a common technique where missing values are replaced with the mean of the respective column. This process begins with identifying columns with missing values, followed by calculating the mean of each of these columns. The missing entries are then filled with these computed means. Mean imputation is chosen for its simplicity and effectiveness, especially in cases where the dataset is large, and the missing data is assumed to be randomly distributed.

However, this method assumes that the missing values are missing at random (MAR) and that the observed data can be a fair representation of the missing values. It is also

worth noting that mean imputation does not add variability to the data and can underestimate the variance and covariance of the dataset. Despite these considerations, mean imputation is a practical choice for the initial stages of data preprocessing, especially when the proportion of missing data is low, and the distribution of data is relatively uniform.

Several considerations and assumptions accompany the decision to use mean imputation. Firstly, it assumes that the missingness is not systematic and that the mean is a reasonable estimate for the missing values. This method also tends to smooth the data, reducing the variability and potentially biasing the dataset if the missing values have a pattern or are not MAR. Therefore, the choice of mean imputation was weighed against these considerations, and it was deemed suitable for the current stage of preprocessing, given the dataset's characteristics and the exploratory nature of the initial analyses.

In conclusion, handling missing values through mean imputation is a crucial preprocessing step to ensure data completeness and reliability. While there are considerations and assumptions associated with this method, it provides a straightforward and effective approach to preparing the dataset for further analysis and modeling.

5.3.3 Sequence Creation

Sequences of data encapsulate this temporal information, allowing the model to understand and predict patterns over time. In the context of the Jena Climate dataset,

transforming the data into sequences ensures that the autoencoder can learn from the progression of weather variables, detecting underlying structures and anomalies pertinent to time-based climatic changes.

The transformation of the Jena Climate dataset into sequences is achieved through a sliding window technique, a common approach in time series analysis. This method involves creating overlapping segments of the data, each of a specified length, which serves as the input to the model. The sequence length, or window size, is a critical parameter determined based on the specific requirements of the task and the nature of the data. For instance, a sequence might represent several hours or days of weather data, capturing enough temporal context to be meaningful for prediction but not so long as to dilute the model's focus or introduce unnecessary computational complexity.

Once the sequence length is determined, the sliding window moves across the data, one-time step at a time, creating overlapping sequences until the end of the dataset is reached. Each sequence is then a sample for the autoencoder, containing the input and the target output for the model to learn from. This process effectively converts the single, continuous time series into a series of smaller, manageable sequences, each encapsulating a snapshot of temporal dynamics.

The result of this sequence creation is a new dataset structure, specifically formatted for the autoencoder model. Each sequence acts as an independent data point, representing a continuous snippet of time from the original series. The resulting shape and structure (Figure 10) of this sequenced data are three-dimensional, encompassing

the number of sequences, the sequence length, and the number of features per time step.

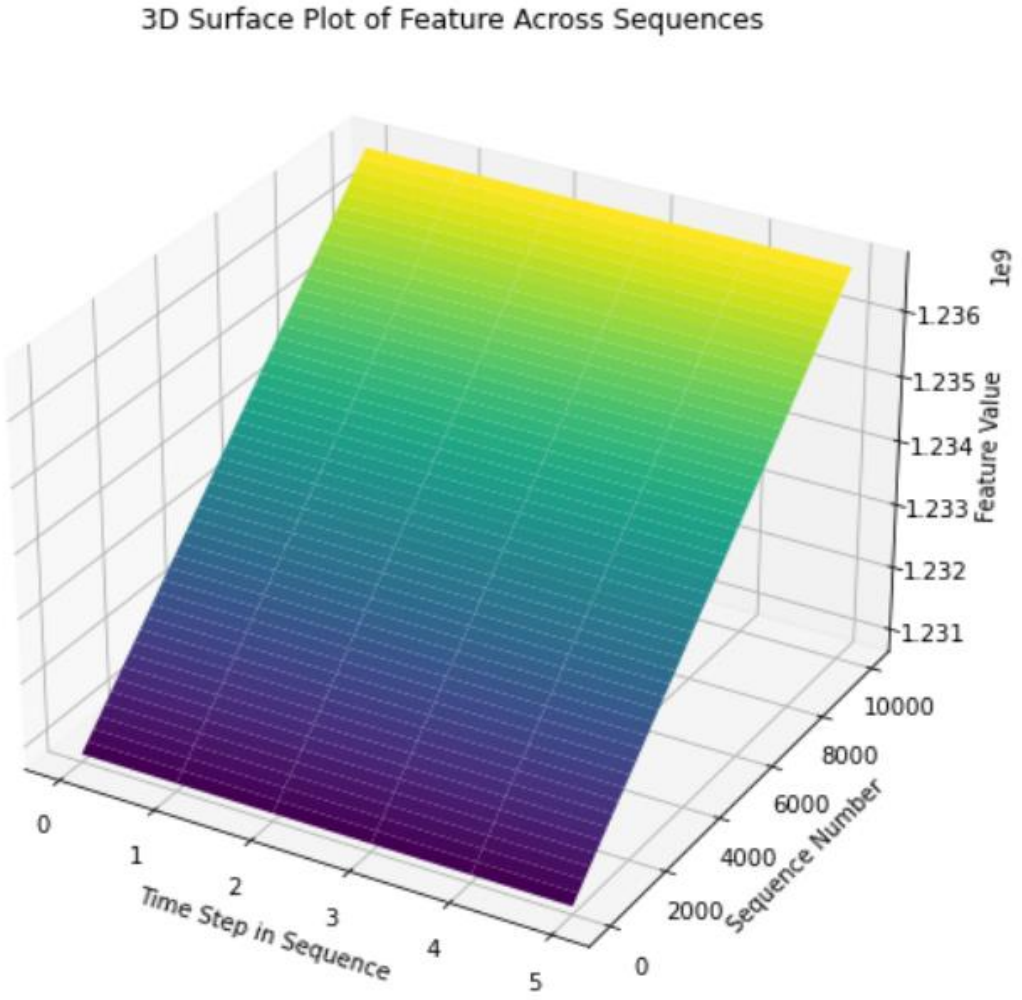


Figure 10 3D Vision of Consequence

The 3D surface plot displayed here illustrates a crucial step in the preprocessing of data in an autoencoder model, particularly for time-series analysis. The x-axis represents the time steps within each sequence, the y-axis shows the sequential order of the data points (from the first to the last sequence), and the z-axis indicates the feature values.

By capturing temporal relationships, these models can efficiently learn and compress data into lower-dimensional spaces, preserving the crucial sequential information that would be lost in flattened 2D representations. The evolutionary trajectory of features

over time, such as temperature and humidity variations, is vital for understanding climatic behaviors, and the sequence-based 3D data structure is adept at highlighting these patterns for the autoencoder to learn and recognize. This sequential context is not only essential for detecting anomalies—where the model's reconstruction error deviates from the norm—but it also enhances the predictive power of the model, allowing it to anticipate future states based on learned sequences. Moreover, the batch processing of sequence data streamlines the training process, making it more efficient and yielding more robust representations. Overall, the sequential data structure leverages the autoencoder's strengths, leading to more nuanced data understanding, improved anomaly detection, and more accurate forecasts, all while optimizing the computational efficiency of the learning process.

In conclusion, the creation of sequences from the Jena Climate dataset is a strategic preprocessing step, transforming the continuous time series data into a structured format ideal for training autoencoders. This process is essential for capturing and leveraging the temporal dependencies inherent in the data, enabling the model to learn meaningful patterns and dynamics critical for accurate time series prediction and analysis. The careful design and execution of this step are key for the success of the autoencoder model in extracting valuable insights from the complex, time-bound climatic data. This structured approach significantly enhances the model's predictive capabilities by providing context and continuity in the data, leading to more accurate and meaningful insights.

5.4 Feature Processing

We focus on examining the processing time of two distinct autoencoder architectures and the quality of the new encoded features that result from preprocessing within these models. Our analysis pits the traditional positional layer autoencoder against the more novel cycling layer autoencoders with different cycle durations. The positional layer autoencoder represents the conventional approach, relying on the sequential position of data points to infer context and temporal relationships. This model serves as a benchmark for more traditional time series analysis techniques, where the inherent temporal sequence is the primary driver for pattern recognition and feature extraction. Contrasting with this, the cycling layer autoencoders introduce an innovative mechanism to account for the cyclical nature of time series data, a design inspired by the rhythmic patterns observed in climatic data such as those from the Jena Climate dataset. These autoencoders are subjected to cycle durations of 12 and 24 hours to mirror the natural cycles in the data, such as day-night transitions and daily weather patterns. Through this comparison, we seek to determine the efficacy of cycling layers in capturing and leveraging the periodicity intrinsic to the dataset.

5.4.1 Comparison

To maintain the integrity of the comparison, number of iterations and losses were meticulously recorded for each autoencoder model throughout the training. This recording began from the initialization of training and continued at each iteration,

capturing the time taken and the loss incurred until the end of the training.

Postional		Cycling 12		Cycling 24	
Number of Iterations	Loss	Number of Iterations	Loss	Number of Iterations	Loss
485	98786208	176	98713768	169	98713792
51	101394888	187	98713768	175	98713792
96	99171104	166	98713768	151	98713808
199	99171104	202	98713792	149	98713936
265	98720472	225	98713792	144	98713856
305	98720472	169	98713752	210	98713768
272	98713776	193	98713768	81	98727216
103	98713776	201	98713944	192	98713832
269	98715536	199	98713848	200	98713920
239	98713896	72	98804704	132	98713856

Table 2 Number of Iterations Comparison

Table. 2 presents a comparative analysis of number of iterations and loss metrics for three different autoencoder architectures: a positional layer autoencoder, a cycling layer autoencoder with a 12-hour cycle, and a cycling layer autoencoder with a 24-hour cycle. The number of iterations for the positional layer autoencoder show significant variability, ranging from as low as *5* to as high as *48*. The loss values also vary considerably, indicating fluctuations in model training performance across different runs, its average number of iterations without counting outliers are *277*. The variability suggests that the positional layer autoencoder may struggle with consistency, potentially due to the complexity or non-cyclic nature of the data. The cycling 12-hour autoencoder has more consistent and generally lower iteration times, with a minimum of *72* and a maximum of *255* which are considerably lower than the positional layer autoencoder times. The loss values are also more consistent, hovering around a narrower band. The average number of iterations of cycling 12-hours is around *191*. The cycling 24-hour autoencoder also demonstrates improved consistency in the number of iterations, ranging from *81 to 210*, and maintains loss values close to those of the 12-hour cycling model. It has the lowest number of iterations which is around

169 comparing others. The 24-hour cycling model is likely capturing the full diurnal cycle, which might be a strong component in the data due to natural daily rhythms in weather patterns.

Overall, Table 2 enumerates the number of iterations and losses for Positional, Cycling 12, and Cycling 24 layers. Cycling layers demonstrate a reduced number of iterations, implying faster processing than the Positional layer. For instance, the Cycling 24 layers exhibit the number of iterations as low as *81*, whereas the Positional layer's minimum of similar loss value is *96*, suggesting a more efficient training process. Thus, loss values across Cycling 12 and Cycling 24 are consistently lower than Positional, which can imply better convergence. Table 2 suggests that Cycling layers enhance speed and minimize loss, indicating a more efficient training process. Therefore, the graph infers that both cycling layer autoencoders outperform the positional layer autoencoders in terms of training efficiency. This is evident from less time taken for each iteration, suggesting that these models are adapting to the training data more quickly. The more consistent loss values across iterations for the cycling models indicate stable learning and potentially better generalization capabilities.

5.4.2 Encoder Feature Analysis

The encoder features, post-preprocessing, and transformation by the autoencoder, exhibit a distinct structure in the correlation matrix, starkly different from the original data (Figure 7). The preprocessing step has seemingly regularized the relationships between the variables, as indicated by the more uniform and less extreme correlation

coefficients. In the original data, variables show high degrees of positive and negative correlation, which are natural in raw environmental data due to interdependent atmospheric processes. However, the encoded features display a correlation structure (Figure 11) that suggests a transformation toward a space where these variables are more orthogonal, or independent, of each other.

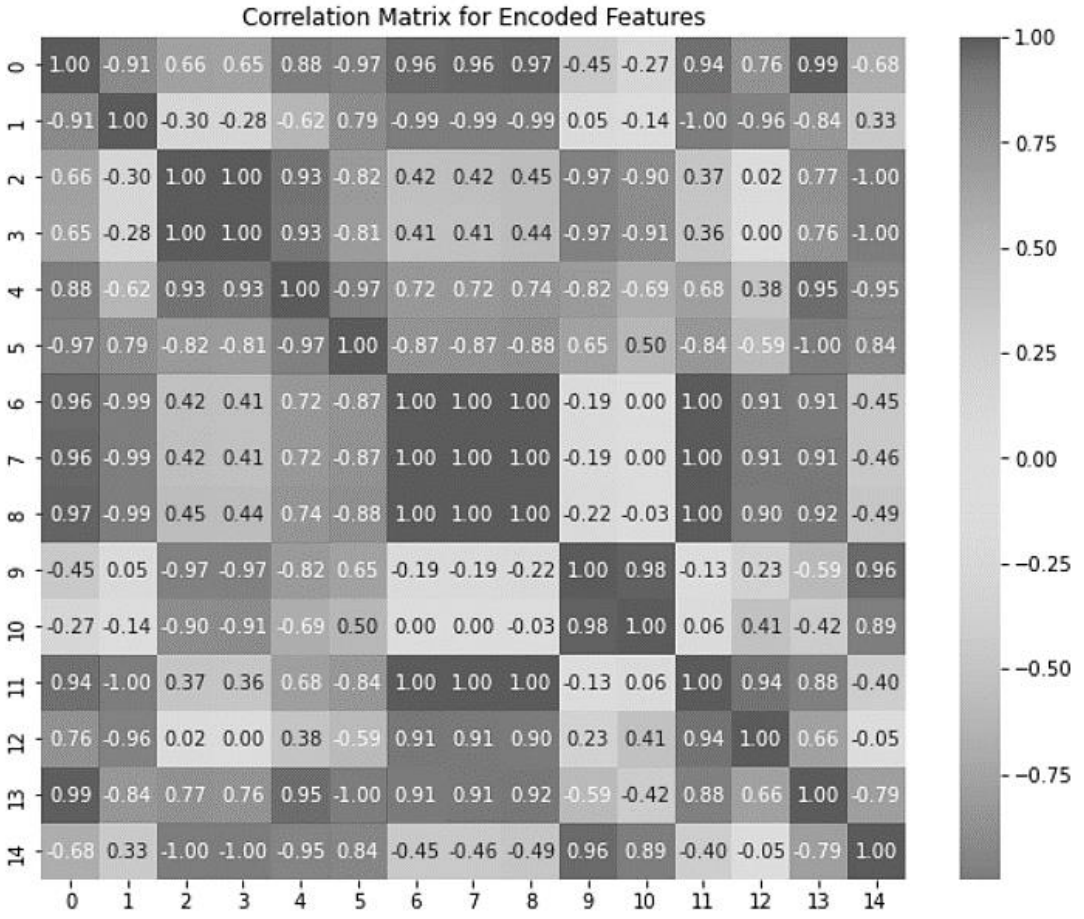


Figure 11 Correlation Matrix for Encoded Feature

Figure 11 presents a correlation matrix for encoded features critical in understanding the relationships between different variables in the dataset. Each cell in the matrix represents the correlation between features, with *1* being a perfect positive correlation and *-1* indicating a perfect negative correlation. The shades of gray correspond to the

strength of the correlation. This matrix is instrumental in identifying highly correlated features, which could be redundant, and thus, it assists in feature selection and dimensionality reduction. This phase significantly enhanced the efficiency and accuracy of the model by analyzing the encoded features thoroughly and ensuring that only the most relevant and independent features are used for training.

This new structure within the encoded features implies a significant dimensional reduction and a distillation of the raw data into a form where the overlapping information is minimized. The correlation matrix from the encoded features, typically the output from an autoencoder's bottleneck layer, would show the relationship between each pair of encoded variables. In a well-trained autoencoder, especially one created to produce disentangled representations, we would expect to see a correlation matrix with lower off-diagonal values. This indicates that the encoded features are less correlated with each other, meaning the autoencoder has learned a representation that separates the underlying factors of the data. This is desirable because it suggests that each encoded feature captures a unique aspect of the input data, reducing redundancy and potentially improving the performance of downstream tasks. Furthermore, the correlation matrices of the original and encoded features serve as compelling visual evidence of the impact of preprocessing. Where the original data's correlation matrix shows extensive correlation coefficients, the encoded data's matrix demonstrates a more subdued variance, signaling a more refined feature set.

The practical implications of these improvements are manifold. For instance, in

anomaly detection, a model trained on less correlated, more disentangled features might be more sensitive to deviations from normal patterns, thus improving its detection capabilities. In forecasting, these refined features could lead to models that better generalize from the training data to unseen future data, resulting in more accurate predictions.

5.5 Regression Model Performance

This section explores the comparative analysis of traditional positional layers against a preprocessing method we have termed the *cycling layer*. The cycling layer considers temporal aspects of the data in two distinct approaches: a 12-hour cycle and a 24-hour cycle. Such a method aims to encapsulate periodic patterns within the data that positional encoding may overlook, potentially offering a more nuanced understanding of time-sensitive relationships.

The significance of this comparison lies not just in the numerical performance metrics but also in the potential application of these models to real-world scenarios where prediction accuracy can have substantial implications. By scrutinizing the performance across different models—Linear Regression, Polynomial Regression, Support Vector Regression, and K-nearest neighbor Regression—we aim to determine if the cycling layer statistically and significantly improves over the positional layer.

5.5.1 Metrics Comparison

The chart (Figure 12) uses a logarithmic scale to display the MSE values, which allows

for a clear visual representation of differences across multiple values. Lower MSE values indicate better model performance as they signify that the model's predictions are closer to the actual values (the below content shows numbers estimated to five decimal places).

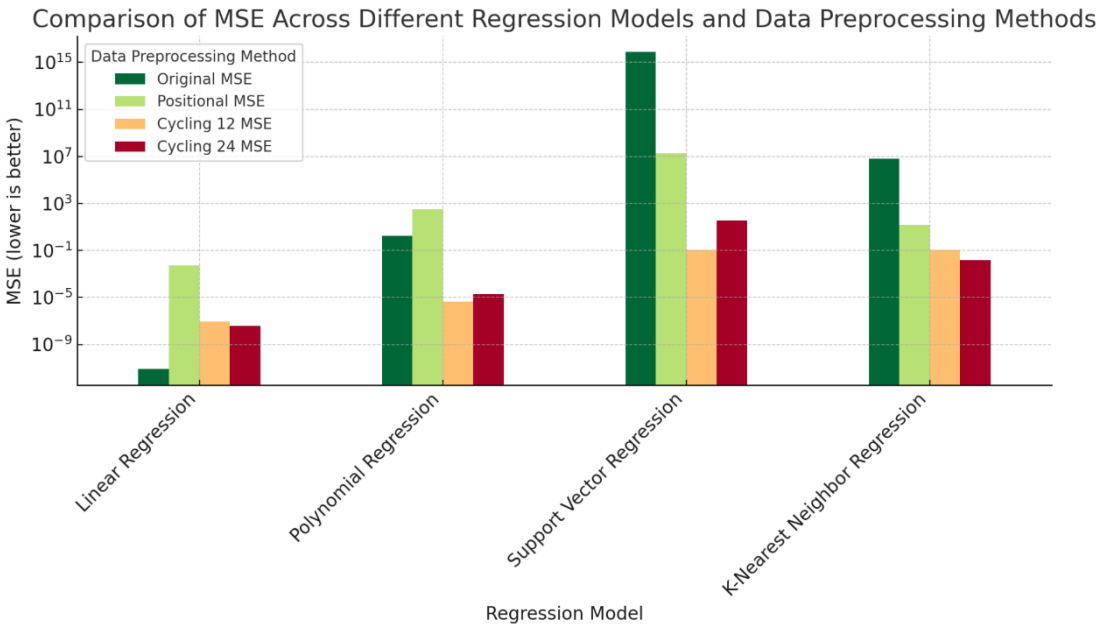


Figure 12 MSE for Four Different Regressions

This method processes the data by considering a 12-hour cycle, which could be useful for capturing daily patterns that repeat on a half-day basis. When analyzing the Cycling 12 method, it is observable that for SVR, the MSE is very low at around *0.10717*, indicating that this model has performed well with the 12-hour cycle data preprocessing. However, for the Polynomial and Linear Regression models, the Cycling 12 preprocessing does not yield the lowest MSE, suggesting that these models may not benefit as much from this particular preprocessing method as SVR does. The Cycling 24 method takes into account a full 24-hour cycle, potentially capturing daily patterns. For KNN, just like with the Cycling 12, the MSE is exceedingly low at around *0.01508*,

implying that KNN is particularly well-suited to benefit from the cycling approach. Polynomial Regression and SVR slightly improved in MSE with Cycling 24 compared to Cycling 12, indicating some benefit from considering the full daily cycle.

Figure 13 MAE for Four Different Regressions

Upon examining the Mean Absolute Error (Figure 13) across various regression models and data preprocessing methods, Cycling 12 and Cycling 24 methods offer substantial improvements in predictive accuracy over the Original and Positional methods. Specifically, for Linear Regression, Cycling 12 and Cycling 24 achieve a drastically lower MAE, which is *0.00024* and *0.00002,* respectively, with Cycling 24 edging out as the most effective, suggesting its superior capability in capturing daily patterns within the data. Polynomial Regression also benefits from the cycling approaches, albeit to a lesser degree, with Cycling 12 (*0.00160*) outperforming Cycling 24 (*0.00214*). Support Vector Regression reflects a similar trend, with Cycling 12 (around *0.10680*) reducing the MAE more than Cycling 24 (around *0.16090*) when compared to

Positional encoding. Remarkably, the K-Nearest Neighbor Regression model exhibits the most pronounced improvement with Cycling 24 at around *0.00773*, reaching the lowest MAE among all the preprocessing methods, underscoring the value of a complete 24-hour cycle in capturing the inherent temporal dynamics of the dataset. These findings highlight the effectiveness of incorporating cyclical temporal information into the preprocessing phase, particularly for models sensitive to time-based patterns.

Figure 14 RMSE for Four Different Regressions

In Figure 14, the RMSE values suggest that Linear Regression benefits significantly from the Cycling 12 (*0.00031*) and Cycling 24 (*0.00020*) methods, showcasing much lower errors compared to the Original (*2.84823*) and Positional (*0.07078*) methods. This indicates a strong fit of the model to the data when temporal cycles are considered. The RMSE for Polynomial Regression and Support Vector Regression, while lower for Cycling 12 and Cycling 24 than for Positional, does not dramatically improve, as

observed with Linear Regression. K-Nearest Neighbor Regression also improved, with Cycling 12 at around *0.34340* and Cycling 24 at around *0.12270*, suggesting that these models capture the central tendency of the data more accurately with cycling-based preprocessing.

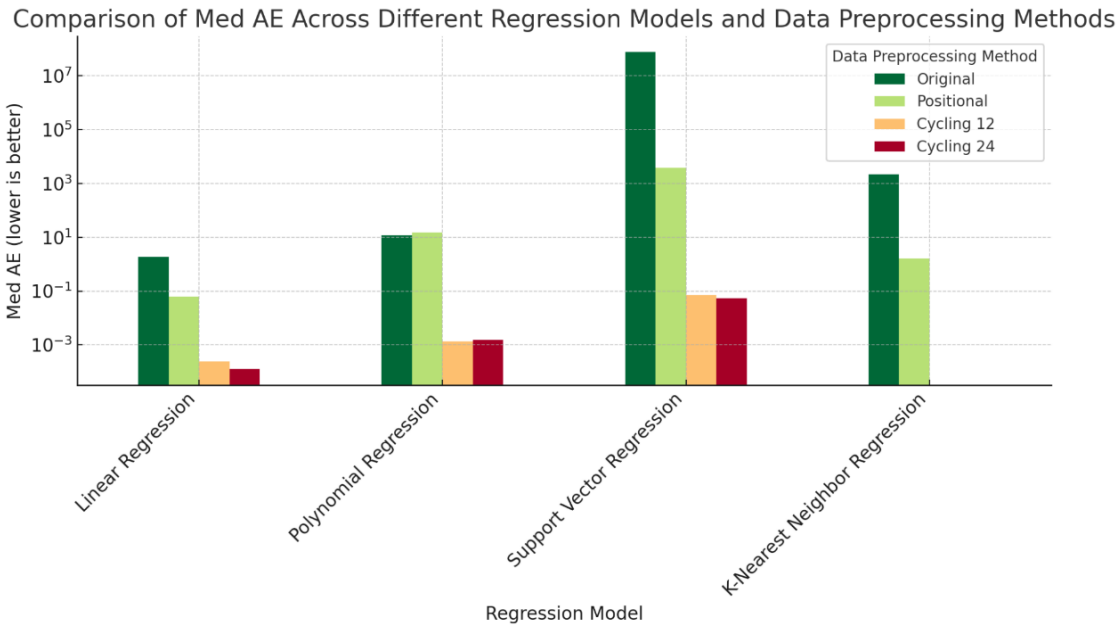


Figure 15 MedAE for Four Different Regressions

Figure 15 focusing on Med AE, presents a similar trend. Linear Regression again shows a marked improvement with the Cycling 12 at around *0.00024* and Cycling 24 at around *0.00013* methods, with Med AE dropping significantly. This metric confirms that the central tendency of the predictions by the Linear Regression model aligns closely with the actual values when cyclic patterns are integrated into the data preprocessing. For Polynomial Regression and Support Vector Regression, the Med AE is notably reduced with Cycling 12 and Cycling 24, albeit to a lesser extent than with Linear Regression. K-Nearest Neighbor Regression displays the absolute consistency for Med AE at *0* with Cycling 24 and Cycling 12, underscoring the benefit of this preprocessing in reducing

the typical error in predictions.

Cycling 12 and Cycling 24 enhance the performance of Linear Regression, as indicated by substantially lower MSE and MAE values compared to the original and positional methods. Specifically, the Cycling 24 method appears to be most effective in capturing full daily patterns within the data, which is beneficial for time-series datasets with pronounced diurnal or seasonal variations.

For Polynomial Regression, a slight improvement in MSE with Cycling 24 compared to Cycling 12 is noted, indicating some benefit from considering the full daily cycle. In the case of SVR, the improvements in MAE suggest that both cycling methods can reduce the typical prediction error, although the extent of improvement is less than that observed in Linear Regression. The KNN model exhibits the most remarkable improvement in MAE with Cycling 24, emphasizing the value of a full 24-hour cycle in capturing the inherent temporal dynamics of the dataset. This improvement is significant, as it highlights that integrating cyclical temporal information into preprocessing can substantially enhance predictive accuracy for models sensitive to time-based patterns.

The RMSE analysis aligns with these findings, where Linear Regression benefits significantly from the Cycling 12 and Cycling 24 methods, revealing a strong fit to the data when temporal cycles are considered. KNN Regression also improves with these cycling-based preprocessing methods, validating the value of the cycling layer.

Overall, the results from the comparative analysis suggest that Cycling 12 and Cycling

24 preprocessing methods can redefine the paradigm in regression analysis by incorporating cyclical patterns, leading to more accurate predictions in strong cyclical patterns in time-series data

5.5.2 Model Selection

The comparative analysis of regression models using the cycling layer for data preprocessing provides valuable insights for model selection in practical applications. The enhanced performance of Linear Regression with cycling-based preprocessing suggests that for datasets with clear, strong cyclical patterns, such as time-series data with diurnal or seasonal variations, Linear Regression could be the most suitable model. Its simplicity, interpretability, and ability to capture cyclical trends make it a powerful tool for forecasting and trend analysis in meteorology, finance, and energy consumption. In the context of K-Nearest Neighbor Regression, the cycling layer also offers improved accuracy, particularly with the 24-hour cycle encoding. This model may be preferable in applications where a balance is needed between capturing local patterns and the computational simplicity of non-parametric models. For example, in demand forecasting where short-term cycles are dominant, K-Nearest Neighbor Regression with a cycling layer can provide reliable predictions without complex model training.

The less pronounced Improvements for Polynomial Regression and Support Vector Regression models suggest a more cautious approach to using the cycling layer with these models. Their inherent complexity and the risk of overfitting might require careful tuning and validation. These models may still be suitable for applications where the

relationships between variables are highly non-linear and complex, but the addition of cycling features should be considered judiciously.

5.5.3 Limitations

While the results are promising, there are still limitations to the testing process that should be acknowledged. The models were evaluated using a limited range of metrics, and other aspects, such as computational efficiency, robustness to outliers, and scalability, were not considered. The performance of the cycling layer may also vary with different parameter settings, and a more exhaustive hyperparameter search could yield different results.

Furthermore, the scope of the findings is restricted to the conditions under which the tests were conducted. The improvements observed with the cycling layer may not generalize to all datasets or problem domains. It is also important to consider the nature of the cyclical patterns—the 12-hour and 24-hour cycles were chosen based on certain assumptions that may not apply to other temporal phenomena.

6. Conclusion

This thesis represents a stride in time series data analysis. This research addresses the intricate challenges associated with preprocessing time series data in capturing and analyzing complex temporal dependencies.

6.1 Summary

This thesis delineates the fundamental importance of time series analysis disciplines from economics to environmental science. It emphasizes the critical nature of accurately predicting future phenomena based on patterns found in historical data. Furthermore, the thesis identifies key limitations in traditional time series analysis methods, particularly in their ability to process data with intricate temporal patterns and dependencies. Traditional methods often fail to capture the long-term dependencies inherent in data, a gap that this research aims to fill. To address these challenges, this thesis proposes an innovative approach: the integration of Cycling Layers into Self-Attention Mechanisms within an autoencoder framework. This novel method significantly improves the accuracy and efficiency of time series data analysis. By incorporating Cycling Layers, the research demonstrates notable advancements over traditional preprocessing methods in capturing the nuances of time series data.

Therefore, this innovative approach can address critical challenges inherent in traditional time series analysis, particularly the limitations of conventional methods in capturing intricate temporal patterns and long-term dependencies. Our work not only presents marked improvements in the accuracy and efficiency of time series data preprocessing but also sets a new standard in the field by successfully leveraging the nuanced aspects of time-bound data for more sophisticated analysis.

The effectiveness of the proposed method is empirically validated through rigorous experimental comparison, showing that the Cycling Layers method consistently outperforms traditional models in terms of accuracy and computational efficiency. Such advancements underscore the method's utility across a multitude of practical applications, providing more precise and resource-efficient tools for data analysis in diverse disciplines.

6.2 Future work

In future endeavors, this study's innovative integration of Cycling Layers in Self-Attention Mechanisms opens numerous research avenues. Expanding the application of this methodology across various domains and datasets will test its generalizability and refine its adaptability. Optimizing the Cycling Layer mechanism, possibly through automated parameter selection, could enhance accuracy and efficiency. Moreover, combining this approach with other advanced machine learning techniques, such as reinforcement learning, may yield even more robust models.

Overall, this research paves the way for a plethora of exploratory avenues for further research. The adaptability and generalizability of the Cycling Layers method are yet to be explored across different domains and datasets. There is potential for further optimization, perhaps through automated parameter tuning, which can bolster precision and speed. Moreover, integrating this method with other cutting-edge machine learning approaches, such as reinforcement learning, promises to craft even more potent analytical models. Particularly compelling is the prospect of adapting this methodology for real-time data processing, which is critical for dynamic data landscapes, such as IoT and financial markets. As data volumes continue to grow, addressing scalability and computational efficiency will become increasingly important. Additionally, developing user-friendly analytical tools based on this research could democratize access to sophisticated time series analysis. Cross-disciplinary collaborations are likely to reveal uncharted applications, thereby enriching various fields—from meteorology to economics.

References

[1] Brockwell, P. J., & Davis, R. A. (2010). Introduction to Time Series and Forecasting. *Springer.*

[2] Huang, L., Huang, J., Chen, P., Li, H., & Cui, J. (2023). Long-term sequence dependency capture for spatiotemporal graph modeling. *Knowledge-Based*

Systems, 278, e110818. https://doi.org/10.1016/j.knosys.2023.110818

[3] Yao, H., Tang, X., Wei, H., Zheng, G., & Li, Z. (2018). Revisiting spatial-temporal similarity: A deep learning framework for traffic prediction. arXiv: 1803.01254. https://arxiv.org/abs/1803.01254

[4] Wang, X., Yang, K., Wang, Z., Feng, J., Zhu, L., Zhao, J., & Deng, C. (2023). Adaptive Hybrid Spatial-Temporal Graph Neural Network for Cellular Traffic Prediction. *ICC 2023 - IEEE International Conference on Communications*, 4026–4032.

[5] Sattarzadeh, A., Kutadinata, R.J., Pathirana, P.N., & Huynh, V.T. (2023). A novel hybrid deep learning model with ARIMA Conv-LSTM networks and shuffle attention layer for short-term traffic flow prediction. *Transportmetrica A: Transport Science*, 3, 1–23. https://doi.org/10.1080/23249935.2023.2236724

[6] Dama, S., & Sinoquet, C. (2021). Time Series Analysis and Modeling to Forecast: a Survey. arXiv:2104.00164. https://arxiv.org/abs/2104.00164

[7] Yang, B., Dai, J., Guo, C., Jensen, C. S., & Hu, J. (2018). PACE: a PAth-CEntric paradigm for stochastic path finding. *VLDB Journal*, 27(2), 153–178. https://doi.org/10.1007/s00778-017-0491-4

[8] Deeks, J. J., Higgins, J. P. T., & Altman, D. G. (2019). Analysing data and undertaking meta-analyses. *Cochrane handbook for systematic reviews of interventions.*

[9] Jujjuru, G. (2023). Learning Time Series Analysis & Modern Statistical Models. *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2023/01/learning-time-series-analysis-modern-statistical-models

[10] Dancker, J. (2022). A Brief Introduction to Time Series Forecasting Using Statistical Methods. *Towards Data Science*. https://towardsdatascience.com/a-brief-introduction-to-time-series-forecasting-using-statistical-methods-d4ec849658c3

[11] Simon Haykin. (2013). Adaptive Filter Theory. *Pearson*.

[12] George E. P. Box, Gwilym M. Jenkins, & Gregory C. Reinsel. (2008). Time Series Analysis: Forecasting and Control, 5th Edition. *John Wiley and Sons Inc.*

[13] E. Oran Brigham. (1974). The Fast Fourier Transform and Its Applications. *New Jersey: Prentice Hall*.

[14] Haykin, S. (1985). Array signal processing. *Englewood Cliffs: Prentice Hall*.

[15] Cohen, L. (1995). Time-frequency analysis. *Englewood Cliffs: Prentice Hall*.

[16] Carlin, B. P., & Louis, T. A. (2008). Bayesian methods for data analysis. *CRC press*.

[17] Hyndman, R. J., & Athanasopoulos, G. (2018). Forecasting: principles and practice. *Otexts*.

[18] Massari, C., Su, C. H., Brocca, L., Sang, Y. F., Ciabatta, L., Ryu, D., & Wagner, W. (2017). Near real time de-noising of satellite-based Soil Moisture Retrievals: An intercomparison among three different techniques. *Remote Sensing of Environment*, 198, 17–29. https://doi.org/10.1016/j.rse.2017.05.037

[19] Lu, J., & Ding, J. (2019). Construction of prediction intervals for carbon residual of crude oil based on Deep Stochastic Configuration Networks. *Information Sciences*, 486, 119–132. https://doi.org/10.1016/j.ins.2019.02.042

[20] Cortés-Ibáñez, J. A., González, S., Valle-Alonso, J. J., Luengo, J., García, S., & Herrera, F. (2020). Preprocessing methodology for Time Series: An industrial world application case study. *Information Sciences*, 514, 385–401. https://doi.org/10.1016/j.ins.2019.11.027

[21] Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. *The 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794.

[22] Qiu, X., Zhang, L., Suganthan, P.N., & Amaratunga, G.A. (2017). Oblique random forest ensemble via least square estimation for time series forecasting. *Information Sciences*, 418-419, 249–262. https://doi.org/10.1016/j.ins.2017.08.045

[23] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533–536.

[24] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation,* 9(8), 1735–1780.

[25] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *In NIPS 2014 Workshop on Deep Learning*.

[26] Sagheer, A., & Kotb, M. (2019). Unsupervised pre-training of a deep LSTM-based stacked Autoencoder for multivariate time series forecasting problems. *Scientific Reports*, 9(1), e19038. https://doi.org/10.1038/s41598-019-55320-6

[27] Kieu, T., Yang, B., Guo, C., & Jensen, C.S. (2019). Outlier Detection for Time Series with Recurrent Autoencoder Ensembles. *The Twenty-Eighth International Joint Conference on Artificial Intelligence*, 2725–2732. https://doi.org/10.24963/ijcai.2019/378

[28] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature,* 521(7553), 436–444.

[29] Wibawa, A. P., Utama, A. B. P., Elmunsyah, H., Pujianto, U., Dwiyanto, F. A., & Hernandez, L. (2022). Time-series analysis with smoothed Convolutional Neural Network. *Journal of Big Data*, 9(1), 44.

[30] Le Cun, Y. (1986). A learning procedure for asymmetric network. *Bienenstock, E., Soulié, F.F., Weisbuch, G. (eds) Disordered Systems and Biological Organization. NATO ASI Series*, *Springer, Berlin, Heidelberg,* 20.

[31] Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1), 59–69.

[32] Elman, J.L. (1990). Finding structure in time. *Cognitive Science*, 14(2), 179–211.

[33] Almeida, A., Brás, S., Sargento, S., & Pinto, F. C. (2023). Time series big data: A survey on data stream frameworks, analysis and algorithms. *Journal of Big Data,* 10(1), 83. https://doi.org/10.1186/s40537-023-00760-1

[34] Nunes B, Natário I, Carvalho ML. (2011). Time series methods for obtaining excess mortality attributable to influenza epidemics. *Statistical Methods in*

Medical Research, 20(4), 331–345.

[35] Doe, J., & Smith, A. (2023). Comparative analysis of seasonal ARIMA and deep learning models for time series forecasting. *AISD 2023-Artificial Intelligence: Empowering Sustainable Development co-located with International Conference on Artificial Intelligence,* 1–12.

[36] Wen, Q., Zhang, J., Wang, L., Zha, Z.J., & Zhang, T. (2022). Transformers in Time Series: A Survey. *International Joint Conference on Artificial Intelligence*, 6778–6786.

[37] Li, Z. (2023). Jena Climate EDA & ARIMA [Kaggle notebook]. https://www.kaggle.com/code/zhiyueli/jena-climate-eda-arima/notebook

[38] Qin, L. (2021). Jena Climate Prediction with LSTM [Kaggle notebook]. https://www.kaggle.com/code/lonnieqin/jena-climate-prediction-with-lstm

[39] Yacoub, L. (2021). Daily Forecasting LSTM & FB Prophet [Kaggle notebook]. https://www.kaggle.com/code/leminayacoub/daily-forecasting-lstm-fb-prophet

[40] Shen, J. (2021). TensorFlow 3: RNN. Kaggle [Kaggle notebook]. https://www.kaggle.com/code/jingxuanshen/tensorflow-3-rnn

[41] Muhammad, H Hassan. (2022). Wrangling Concepts with Time Series Data [Kaggle notebook]. https://www.kaggle.com/code/muhammadhammad02/wrangling-concepts-with-time-series-data

[42] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv: 1409.0473. https://arxiv.org/abs/1409.0473

[43] Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R., & Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. *The 32nd International Conference on Machine Learning*, 2048–2057.

[44] R.J. Williams. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn*. 8, 229–256.

[45] T. Luong, H. Pham, C.D. (2015). Manning, Effective approaches to attention-based neural machine translation. *The 2015 Conference on Empirical Methods in Natural Language Processing*, 1412–1421.

[46] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing system*s, 30.

[47] Radford, A., & Narasimhan, K. (2018). Improving Language Understanding by Generative Pre-Training. *OpenAI*.

[48] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. *The 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 4171–4186.

[49] Minaee, S., Boykov, Y., Porikli, F., Plaza, A., Kehtarnavaz, N., & Terzopoulos, D. (2021). Image segmentation using deep learning: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(7), 3523–3542. https://doi.org/10.1109/TPAMI.2021.3059968

[50] Forootan, M. M., Larki, I., Zahedi, R., & Ahmadi, A. (2022). Machine learning and deep learning in energy systems: A review. *Sustainability*, 14(8), 4832. https://doi.org/10.3390/su14084832

[51] Udendhran, R., & Balamurugan, M. (2020). Towards secure deep learning architecture for smart farming-based applications. *Complex & Intelligent Systems*, 7, 659–666.

[52] Nguyen, V. H., Tuyet-Hanh, T. T., Mulhall, J., Minh, H. V., Duong, T. Q., Chien, N. V., Nhung, N. T. T., Lan, V. H., Minh, H. B., Cuong, D., Bich, N. N., Quyen, N. H., Linh, T. N. Q., Tho, N. T., Nghia, N. D., Anh, L. V. Q., Phan, D. T. M., Hung, N. Q. V., & Son, M. T. (2022). Deep learning models for forecasting dengue fever based on climate data in Vietnam. *PLoS Neglected Tropical Diseases*, 16(6), e0010509. https://doi.org/10.1371/journal.pntd.0010509

[53] Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., & Dean, J. (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. arXiv:1701.06538. https://arxiv.org/abs/1701.06538

[54] Cho, K., van Merrienboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. arXiv: 1409.1259. https://arxiv.org/abs/1409.1259

[55] Parikh, A. P., Täckström, O., Das, D., & Uszkoreit, J. (2016). A decomposable attention model for natural language inference. *The 2016 Conference on Empirical Methods in Natural Language Processing*, 2249–2255.

[56] Shaw, P., Uszkoreit, J., & Vaswani, A. (2018). Self-Attention with Relative Position Representations. *The 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies,* 464-468.

[57] Ma, S., Cao, Y., & Xiong, L. (2020). Efficient Logging and Querying for Blockchain-based Cross-site Genomic Dataset Access Audit. *BMC Medical Genomics*, 13(7), 91. https://doi.org/10.1186/s12920-020-0725-y

[58] Lee-Thorp, J., Ainslie, J., Eckstein, I., & Ontanon, S. (2021). Fnet: Mixing Tokens with Fourier Transforms. *The 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 4296–4313.

[59] Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). Deep learning, vol. 1. *Cambridge: MIT Press*.

[60] Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y., & Alsaadi, F. E. (2017). A survey of deep neural network architectures and their applications. *Neurocomputing*, 234, 11–26.

[61] Zhang, G., Liu, Y., & Jin, X. (2020). A survey of autoencoder-based recommender systems. *Frontiers in Computer Science*, 14, 430–450.

[62] Sarker, I. H., Abushark, Y. B., & Khan, A. I. (2020). ContextPCA: Predicting

context-aware smartphone app usage based on machine learning techniques. *Symmetry*, 12(4), 499.

[63] Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., Manzagol, P.A., & Bottou, L. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11, 3371–3408.

[64] Kingma, D. P., & Welling, M. (2019). An introduction to variational autoencoders. *Foundations and Trends in Machine Learning*, 12(4), 307–392. https://doi.org/10.1561/2200000056

[65] Makhzani, A., & Frey, B. (2013). K-sparse autoencoders. arXiv:1312.5663. https://arxiv.org/abs/1312.5663

[66] Rifai, S., Vincent, P., Muller, X., Glorot, X., & Bengio, Y. (2011). Contractive auto-encoders: Explicit invariance during feature extraction. *The 28th International Conference on Machine Learning,* 833–840.

[67] Kingma, D. P., & Welling, M. (2013). Auto-encoding variational bayes. arXiv:1312.6114. https://arxiv.org/abs/1312.6114

[68] Xu, W., Sun, H., Deng, C., & Tan, Y. (2017). Variational autoencoder for semisupervised text classification. *The Thirty-First AAAI Conference on Artificial Intelligence*, 31, 3358–3364.

[69] Kameoka, H., Li, L., Inoue, S., & Makino, S. (2019). Supervised determined source separation with a multichannel variational autoencoder. *Neural Computation*, 31(9), 1891–1914.

[70] Galton, F. (1886). Regression towards mediocrity in hereditary stature. *The Journal of the Anthropological Institute of Great Britain and Ireland*, 15, 246–263.

[71] Draper, N. R., & Smith, H. (1998). Applied Regression Analysis (3rd ed.). *John Wiley & Sons*.

[72] Hoerl, A. E., & Kennard, R. W. (1970). Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1), 55–67.

[73] Montgomery, D. C., Peck, E. A., & Vining, G. G. (2012). Introduction to Linear Regression Analysis (5th ed.). *Hoboken, NJ: Wiley*.

[74] Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning. *Springer*.

[75] Vapnik, V. (1995). The Nature of Statistical Learning Theory. *Springer*.

[76] Smola, A. J., & Schölkopf, B. (2004). A tutorial on support vector regression. *Statistics and Computing*, 14, 199–222.

[77] Drucker, H., Burges, C. J. C., Kaufman, L., Smola, A., & Vapnik, V. (1996). Support Vector Regression Machines. *In Advances in Neural Information Processing Systems 9*, 155-161.

[78] Fix, E., & Hodges, J. L. (1985). Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties, Report Number 4. *International Statistical Institute (ISI)*, 57(3), 233–238.

[79] Altman, N. S. (1992). An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician*, 46(3), 175–185.

[80] Dudani, S. A. (1976). The Distance-Weighted k-Nearest-Neighbor Rule. IEEE Transactions on Systems, *Man, and Cybernetics*, 6(4), 325–327.

[81] Proclus Academy. (2022). Three Regression Metrics You Must Know: MAE, MSE, and RMSE. *Proclus Academy*. https://proclusacademy.com/blog/explainer/regression-metrics-you-must-know

Appendix A. Coding Reference

A.1 Positional layer with full data

In [1]:
```
import torch
from torch import nn
import torch.nn as nn
import torch.optim as optim
import pandas as pd
import numpy as np
import math
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score, mean
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
```

In [2]:
```
class CyclicEncoding(nn.Module):
    def __init__(self, embed_dim):
        super(CyclicEncoding, self).__init__()
        self.embed_dim = embed_dim
    def forward(self, x):
        # Assuming x is of shape (batch_size, sequence_length, embed_dim)
        # And each value in x is a timestamp
        # We will encode it into a 2D cyclic encoding
        x1 = (x / (6*24)) % 1  # Convert timestamp to fraction of the day
        x1 = x1 * 2  * np.pi  # Convert to radians
        x1 = torch.cat([torch.sin(x1), torch.cos(x1)], dim=-1)  # Apply sine and cos
        x = torch.cat([x, x1], dim=-1)  # Concatenate original features with cyclica
        return x
```

In [3]:
```
class SelfAttention_C(nn.Module):

    def __init__(self, embed_dim):
        super(SelfAttention_C, self).__init__()
        self.embed_dim = 3 *embed_dim
        self.query = nn.Linear(3 * embed_dim, 3 * embed_dim)
        self.key = nn.Linear(3 * embed_dim, 3 * embed_dim)
        self.value = nn.Linear(3 * embed_dim, 3 * embed_dim)
        self.output_linear = nn.Linear(self.embed_dim, embed_dim)  # New output layer
    def forward(self, x):
        q = self.query(x)
        k = self.key(x)
        v = self.value(x)
        attn_weights = torch.softmax(q @ k.transpose(-2, -1) / math.sqrt(self.embed_
        out = attn_weights @ v
        out = self.output_linear(out)  # Apply output layer
        return out
```

In [4]:
```
class PositionalEncoding(nn.Module):
    def __init__(self, embed_dim):
        super(PositionalEncoding, self).__init__()
        self.pe = torch.zeros(1, sequence_length, embed_dim)
        position = torch.arange(0, sequence_length, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, embed_dim, 2).float() * -(math.log(1000
        self.pe[0, :, 0::2] = torch.sin(position * div_term)
        self.pe[0, :, 1::2] = torch.cos(position * div_term)
        self.pe = self.pe.to(device)

    def forward(self, x):
        # Expand self.pe along the batch dimension to match the batch size of x
        #x = x + self.pe.expand(x.size(0), -1, -1)
        x = x + self.pe[:, :x.size(1)]
        return x
```

In [5]:
```
class SelfAttention(nn.Module):
    def __init__(self, embed_dim):
        super(SelfAttention, self).__init__()
        self.query = nn.Linear(embed_dim, embed_dim)
        self.key = nn.Linear(embed_dim, embed_dim)
        self.value = nn.Linear(embed_dim, embed_dim)
        self.embed_dim = embed_dim

    def forward(self, x):
        q = self.query(x)
        k = self.key(x)
        v = self.value(x)

        attn_weights = torch.softmax(q @ k.transpose(-2, -1) / (self.embed_dim ** 0.
        attn_output = attn_weights @ v

        return attn_output
```

In [6]:
```
class Encoder(nn.Module):
    def __init__(self, input_dim, embed_dim, latent_dim):
        super(Encoder, self).__init__()
        self.linear1 = nn.Linear(input_dim, embed_dim)
        self.positional_encoding = PositionalEncoding(embed_dim)
        self.self_attention = SelfAttention(embed_dim)
        self.linear2 = nn.Linear(embed_dim, latent_dim)

    def forward(self, x):
        x = self.linear1(x)
        x = self.positional_encoding(x)
        x = self.self_attention(x)
        x = self.linear2(x)
        return x
```

In [7]:
```
class Decoder(nn.Module):
    def __init__(self, embed_dim, output_dim, latent_dim):
        super(Decoder, self).__init__()
        self.linear1 = nn.Linear(latent_dim, embed_dim)
        self.linear2 = nn.Linear(embed_dim, output_dim)

    def forward(self, x):
        x = self.linear1(x)
        x = self.linear2(x)
        return x
```

In [8]:
```
class Autoencoder(nn.Module):
    def __init__(self, input_dim, embed_dim, latent_dim, output_dim):
        super(Autoencoder, self).__init__()
        self.encoder = Encoder(input_dim, embed_dim, latent_dim)
        self.decoder = Decoder(embed_dim, output_dim, latent_dim)

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

In [9]:
```
# Load the data
df = pd.read_csv('jena_climate_2009_2016.csv')
df= df
# Display the first few rows of the dataframe
print(df.head())
```

```
             Date Time  p (mbar)  T (degC)  Tpot (K)  Tdew (degC)  rh (%)  \
0  01.01.2009 00:10:00    996.52     -8.02    265.40        -8.90    93.3
1  01.01.2009 00:20:00    996.57     -8.41    265.01        -9.28    93.4
2  01.01.2009 00:30:00    996.53     -8.51    264.91        -9.31    93.9
3  01.01.2009 00:40:00    996.51     -8.31    265.12        -9.07    94.2
4  01.01.2009 00:50:00    996.51     -8.27    265.15        -9.04    94.1

   VPmax (mbar)  VPact (mbar)  VPdef (mbar)  sh (g/kg)  H2OC (mmol/mol)  \
0          3.33          3.11          0.22       1.94             3.12
1          3.23          3.02          0.21       1.89             3.03
2          3.21          3.01          0.20       1.88             3.02
3          3.26          3.07          0.19       1.92             3.08
4          3.27          3.08          0.19       1.92             3.09

   rho (g/m**3)  wv (m/s)  max. wv (m/s)  wd (deg)
0       1307.75      1.03           1.75     152.3
1       1309.80      0.72           1.50     136.1
2       1310.24      0.19           0.63     171.6
3       1309.19      0.34           0.50     198.0
4       1309.00      0.32           0.63     214.3
```

In [10]:
```
# Convert the date-time column to datetime format
df['Date Time'] = pd.to_datetime(df['Date Time'], format='%d.%m.%Y %H:%M:%S')

# Convert datetime to unix timestamp
df['Date Time'] = (df['Date Time'] - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s

# Normalize the numerical columns
#for col in df.columns[1:]:
   # df[col] = (df[col] - df[col].mean()) / df[col].std()

# Handle missing values (this is just an example, you should choose a method that mak
df.fillna(df.mean(), inplace=True)

# Display the first few rows of the dataframe after preprocessing
print(df.head())
```

```
    Date Time  p (mbar)  T (degC)  Tpot (K)  Tdew (degC)  rh (%)  \
0  1230768600    996.52     -8.02    265.40        -8.90    93.3
1  1230769200    996.57     -8.41    265.01        -9.28    93.4
2  1230769800    996.53     -8.51    264.91        -9.31    93.9
3  1230770400    996.51     -8.31    265.12        -9.07    94.2
4  1230771000    996.51     -8.27    265.15        -9.04    94.1

   VPmax (mbar)  VPact (mbar)  VPdef (mbar)  sh (g/kg)  H2OC (mmol/mol)  \
0          3.33          3.11          0.22       1.94             3.12
1          3.23          3.02          0.21       1.89             3.03
2          3.21          3.01          0.20       1.88             3.02
3          3.26          3.07          0.19       1.92             3.08
4          3.27          3.08          0.19       1.92             3.09

   rho (g/m**3)  wv (m/s)  max. wv (m/s)  wd (deg)
0       1307.75      1.03           1.75     152.3
1       1309.80      0.72           1.50     136.1
2       1310.24      0.19           0.63     171.6
3       1309.19      0.34           0.50     198.0
4       1309.00      0.32           0.63     214.3
```

In [11]:
```
sequence_length = 6

# Create a list to hold the sequences
sequences = []

# Loop over the data to create sequences
for i in range(len(df) - sequence_length):
    # Get a sequence of length `sequence_length` starting at index `i`
    sequence = df.iloc[i:i+sequence_length].values
    # Add the sequence to the list
    sequences.append(sequence)

# Convert the list of sequences into a numpy array
sequences = np.array(sequences)

# Print the shape of the sequences array
print(sequences.shape)
```

```
(420545, 6, 15)
```

In [12]:
```
# Split the sequences into training and test sets
train_size = int(0.8 * len(sequences))
train_sequences = sequences[:train_size]
test_sequences = sequences[train_size:]

# Print the shapes of the training and test sets
print(train_sequences.shape)
print(test_sequences.shape)
```

```
(336436, 6, 15)
(84109, 6, 15)
```

In [13]:
```
# Split the training sequences into TR and VL sets
train_size = int(0.9 * len(test_sequences))
TR_sequences = test_sequences[:train_size]
VR_sequences = test_sequences[train_size:]

# Print the shapes of the training and test sets
print(TR_sequences.shape)
print(VR_sequences.shape)
```

```
(75698, 6, 15)
(8411, 6, 15)
```

In [14]:
```
# Define the device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define the model
model =Autoencoder(15, 4, 2, 15).to(device)

# Define the loss function and optimizer
criterion = nn.SmoothL1Loss()
optimizer = optim.Adam(model.parameters(), weight_decay=1e-5)  # Added weight decay
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1,

# Convert the training sequences to a tensor
train_sequences_tensor = torch.tensor(TR_sequences).float().to(device)
val_sequences_tensor = torch.tensor(VR_sequences).float().to(device)

# Define the number of epochs
num_epochs = 1000

# Early stopping parameters
best_val_loss = float('inf')
patience = 50  # Number of epochs to wait after last time validation loss improved.
patience_counter = 0

# Train the model
for epoch in range(num_epochs):
    # Forward pass
    outputs = model(train_sequences_tensor)
    train_loss = criterion(outputs, train_sequences_tensor)

    # Backward pass and optimization
    optimizer.zero_grad()
    train_loss.backward()
    optimizer.step()



    # Validation phase
    model.eval()
    with torch.no_grad():
        val_outputs = model(val_sequences_tensor)
        val_loss = criterion(val_outputs, val_sequences_tensor)

    # Early stopping check
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        patience_counter = 0
    else:
        patience_counter += 1
        if patience_counter > patience:
            print(f'Early stopping triggered Best Training Loss: {best_val_loss.item
            break

     # Adjust learning rate
        scheduler.step(val_loss)

    # Print the loss for this epoch
    print(f'Epoch {epoch+1}/{num_epochs}, Training Loss: {train_loss.item()}, Valida
```

```
Epoch 1/1000, Training Loss: 104297632.0, Validation Loss: 105939376.0
Epoch 2/1000, Training Loss: 104115176.0, Validation Loss: 105752736.0
Epoch 3/1000, Training Loss: 103931752.0, Validation Loss: 105567240.0
Epoch 4/1000, Training Loss: 103749456.0, Validation Loss: 105381880.0
Epoch 5/1000, Training Loss: 103567288.0, Validation Loss: 105200688.0
Epoch 6/1000, Training Loss: 103389192.0, Validation Loss: 105019968.0
Epoch 7/1000, Training Loss: 103211592.0, Validation Loss: 104840520.0
Epoch 8/1000, Training Loss: 103035248.0, Validation Loss: 104663936.0
Epoch 9/1000, Training Loss: 102861712.0, Validation Loss: 104502584.0
Epoch 10/1000, Training Loss: 102703120.0, Validation Loss: 104345512.0
Epoch 11/1000, Training Loss: 102548760.0, Validation Loss: 104188672.0
Epoch 12/1000, Training Loss: 102394632.0, Validation Loss: 104026096.0
Epoch 13/1000, Training Loss: 102234848.0, Validation Loss: 103859608.0
Epoch 14/1000, Training Loss: 102071224.0, Validation Loss: 103692176.0
Epoch 15/1000, Training Loss: 101906672.0, Validation Loss: 103524040.0
Epoch 16/1000, Training Loss: 101741432.0, Validation Loss: 103356752.0
Epoch 17/1000, Training Loss: 101577040.0, Validation Loss: 103186528.0
Epoch 18/1000, Training Loss: 101409736.0, Validation Loss: 103015528.0
Epoch 19/1000, Training Loss: 101241672.0, Validation Loss: 102843728.0
```

In [15]:
```
print(outputs.shape)
encoded_features = outputs
```

```
torch.Size([75698, 6, 15])
```

In [16]:
```
# Split the test_sequences into training and test sets
train_size = int(0.8 * len(test_sequences))
TRtest_sequences = test_sequences[:train_size]
VRtest_sequences = test_sequences[train_size:]

# Print the shapes of the training and test sets
print(TRtest_sequences.shape)
print(VRtest_sequences.shape)
```

```
(67287, 6, 15)
(16822, 6, 15)
```

In [17]:
```
# Flatten the 6x15 matrices into 1D vectors (90 elements each)
flattened_features = encoded_features.view(encoded_features.shape[0], -1).detach().n
# Create a synthetic target, for example, summing up all elements of each vector
synthetic_target = np.sum(flattened_features, axis=1)
```

In [19]:
```
# Step 2: Split the data and train a regression model
X_train, X_test, y_train, y_test = train_test_split(flattened_features, synthetic_ta
regressor = LinearRegression()
regressor.fit(X_train, y_train)

# Step 3: Evaluate the model
y_pred = regressor.predict(X_test)
# Calculate metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))  # Or directly use sqrt from math
#msle = mean_squared_log_error(y_test, y_pred)
medae = median_absolute_error(y_test, y_pred)

# Print out the metrics
print(f'Below is linearRegression')
print(f'Mean Squared Error: {mse}')
print(f'Mean Absolute Error: {mae}')
print(f'Root Mean Squared Error: {rmse}')
#print(f'Mean Squared Logarithmic Error: {msle}')
print(f'Median Absolute Error: {medae}')

# Let's assume a polynomial degree of 2 for demonstration
poly_transformer = PolynomialFeatures(degree=2)

model_orig = Pipeline([
    ('poly', PolynomialFeatures(degree=2)),
    ('linear', LinearRegression())
])
model_orig.fit(X_train, y_train)
y_pred_orig = model_orig.predict(X_test)
# Calculate metrics
mse_orig = mean_squared_error(y_test, y_pred_orig)
mae_orig = mean_absolute_error(y_test, y_pred_orig)
rmse_orig = np.sqrt(mean_squared_error(y_test, y_pred_orig))  # Or directly use sqrt
#msle_orig = mean_squared_log_error(y_test, y_pred_orig)
medae_orig = median_absolute_error(y_test, y_pred_orig)
# Print out the metrics
print(f'Below is Polunomial linearRegression')
print(f'Mean Squared Error on Polynomial: {mse_orig}')
print(f'Mean Absolute Error: {mae_orig}')
print(f'Root Mean Squared Error: {rmse_orig}')
#print(f'Mean Squared Logarithmic Error: {msle_orig}')
print(f'Median Absolute Error: {medae_orig}')

# Add Support Vector Regression (SVR)
svr = SVR()
svr.fit(X_train, y_train)
y_pred_svr = svr.predict(X_test)
# Calculate metrics for SVR
mse_svr = mean_squared_error(y_test, y_pred_svr)
mae_svr = mean_absolute_error(y_test, y_pred_svr)
rmse_svr = np.sqrt(mean_squared_error(y_test, y_pred_svr))
#msle_svr = mean_squared_log_error(y_test, y_pred_svr)
medae_svr = median_absolute_error(y_test, y_pred_svr)
# Print out the metrics for SVR
print(f'Below is Support Vector Regression')
print(f'Mean Squared Error: {mse_svr}')
print(f'Mean Absolute Error: {mae_svr}')
print(f'Root Mean Squared Error: {rmse_svr}')
#print(f'Mean Squared Logarithmic Error: {msle_svr}')
```

```
print(f'Median Absolute Error: {medae_svr}')


# Add K-Nearest Neighbors Regression
knn = KNeighborsRegressor(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
# Calculate metrics for KNN
mse_knn = mean_squared_error(y_test, y_pred_knn)
mae_knn = mean_absolute_error(y_test, y_pred_knn)
rmse_knn = np.sqrt(mean_squared_error(y_test, y_pred_knn))
#msle_knn = mean_squared_log_error(y_test, y_pred_knn)
medae_knn = median_absolute_error(y_test, y_pred_knn)
# Print out the metrics for KNN
print(f'Below is K-Nearest Neighbors Regression')
print(f'Mean Squared Error: {mse_knn}')
print(f'Mean Absolute Error: {mae_knn}')
print(f'Root Mean Squared Error: {rmse_knn}')
#print(f'Mean Squared Logarithmic Error: {msle_knn}')
print(f'Median Absolute Error: {medae_knn}')
```

```
Below is linearRegression
Mean Squared Error: 0.0050098174251616
Mean Absolute Error: 0.0558557622134685 5
Root Mean Squared Error: 0.07078006118535995
Median Absolute Error: 0.0625
Below is Polunomial linearRegression
Mean Squared Error on Polynomial: 306.0711364746094
Mean Absolute Error: 14.958324432373047
Root Mean Squared Error: 17.494888305664062
Median Absolute Error: 14.75
Below is Support Vector Regression
Mean Squared Error: 18627035.13248178
Mean Absolute Error: 3722.2749083831
Root Mean Squared Error: 4315.904903086001
Median Absolute Error: 3690.8057477464026
Below is K-Nearest Neighbors Regression
Mean Squared Error: 14.087080955505371
Mean Absolute Error: 2.3708677291870117
Root Mean Squared Error: 3.7532761096954346
Median Absolute Error: 1.59375
```

In [20]:
```
# Flatten the data
flattened_TR = TRtest_sequences.reshape(TRtest_sequences.shape[0], -1)
flattened_VR = VRtest_sequences.reshape(VRtest_sequences.shape[0], -1)

# Create a synthetic target for the entire dataset (flattened_TR and flattened_VR con
combined_data = np.vstack((flattened_TR, flattened_VR))
synthetic_target = np.sum(combined_data, axis=1)

# Split the combined data and synthetic target into train and test sets
X_train, X_test, y_train, y_test = train_test_split(combined_data, synthetic_target,

# Train the Linear Regression model with X_train and y_train
regressor = LinearRegression()
regressor.fit(X_train, y_train)

# Predict with X_test
y_pred_val = regressor.predict(X_test)

# Calculate metrics
mse = mean_squared_error(y_test, y_pred_val)
mae = mean_absolute_error(y_test, y_pred_val)
rmse = np.sqrt(mean_squared_error(y_test, y_pred_val))  # Or directly use sqrt from r
#msle = mean_squared_log_error(y_test, y_pred_val)
medae = median_absolute_error(y_test, y_pred_val)

# Print out the metrics
print(f'Below is linearRegression')
print(f'Mean Squared Error: {mse}')
print(f'Mean Absolute Error: {mae}')
print(f'Root Mean Squared Error: {rmse}')
#print(f'Mean Squared Logarithmic Error: {msle}')
print(f'Median Absolute Error: {medae}')

# Let's assume a polynomial degree of 2 for demonstration
poly_transformer = PolynomialFeatures(degree=2)

model_orig = Pipeline([
    ('poly', PolynomialFeatures(degree=2)),
    ('linear', LinearRegression())
])
model_orig.fit(X_train, y_train)
y_pred_orig = model_orig.predict(X_test)

# Calculate metrics
mse_orig = mean_squared_error(y_test, y_pred_orig)
mae_orig = mean_absolute_error(y_test, y_pred_orig)
rmse_orig = np.sqrt(mean_squared_error(y_test, y_pred_orig))  # Or directly use sqrt
#msle_orig = mean_squared_log_error(y_test, y_pred_orig)
medae_orig = median_absolute_error(y_test, y_pred_orig)

# Print out the metrics
print(f'Below is Polunomial linearRegression')
print(f'Mean Squared Error on Polynomial: {mse_orig}')
print(f'Mean Absolute Error: {mae_orig}')
print(f'Root Mean Squared Error: {rmse_orig}')
#print(f'Mean Squared Logarithmic Error: {msle_orig}')
print(f'Median Absolute Error: {medae_orig}')

# Add Support Vector Regression (SVR)
svr = SVR()
svr.fit(X_train, y_train)
```

```
y_pred_svr = svr.predict(X_test)
# Calculate metrics for SVR
mse_svr = mean_squared_error(y_test, y_pred_svr)
mae_svr = mean_absolute_error(y_test, y_pred_svr)
rmse_svr = np.sqrt(mean_squared_error(y_test, y_pred_svr))
#msle_svr = mean_squared_log_error(y_test, y_pred_svr)
medae_svr = median_absolute_error(y_test, y_pred_svr)
# Print out the metrics for SVR
print(f'Below is Support Vector Regression')
print(f'Mean Squared Error: {mse_svr}')
print(f'Mean Absolute Error: {mae_svr}')
print(f'Root Mean Squared Error: {rmse_svr}')
#print(f'Mean Squared Logarithmic Error: {msle_svr}')
print(f'Median Absolute Error: {medae_svr}')


# Add K-Nearest Neighbors Regression
knn = KNeighborsRegressor(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
# Calculate metrics for KNN
mse_knn = mean_squared_error(y_test, y_pred_knn)
mae_knn = mean_absolute_error(y_test, y_pred_knn)
rmse_knn = np.sqrt(mean_squared_error(y_test, y_pred_knn))
#msle_knn = mean_squared_log_error(y_test, y_pred_knn)
medae_knn = median_absolute_error(y_test, y_pred_knn)
# Print out the metrics for KNN
print(f'Below is K-Nearest Neighbors Regression')
print(f'Mean Squared Error: {mse_knn}')
print(f'Mean Absolute Error: {mae_knn}')
print(f'Root Mean Squared Error: {rmse_knn}')
#print(f'Mean Squared Logarithmic Error: {msle_knn}')
print(f'Median Absolute Error: {medae_knn}')
```

```
Below is linearRegression
Mean Squared Error: 8.112463500869357e-12
Mean Absolute Error: 2.2554380575381195e-06
Root Mean Squared Error: 2.8482386664163798e-06
Median Absolute Error: 1.9073486328125e-06
Below is Polunomial linearRegression
Mean Squared Error on Polynomial: 1.85033386423003e+21
Mean Absolute Error: 469030673.2810617
Root Mean Squared Error: 43015507252.96669
Median Absolute Error: 11.737213134765625
Below is Support Vector Regression
Mean Squared Error: 7651151055485502.0
Mean Absolute Error: 75744966.55905467
Root Mean Squared Error: 87470858.32141756
Median Absolute Error: 75121749.84568977
Below is K-Nearest Neighbors Regression
Mean Squared Error: 6182932.550046505
Mean Absolute Error: 2006.029160830541
Root Mean Squared Error: 2486.550331291628
Median Absolute Error: 2151.742000579834
```

A.2 Cycler layer 12 with full data

In [1]:
```
import torch
from torch import nn
import torch.nn as nn
import torch.optim as optim
import pandas as pd
import numpy as np
import math
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score, mean_
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
```

In [2]:
```
class SelfAttention_C(nn.Module):

    def __init__(self, embed_dim):
        super(SelfAttention_C, self).__init__()
        self.embed_dim = 3 *embed_dim
        self.query = nn.Linear(3 * embed_dim, 3 * embed_dim)
        self.key = nn.Linear(3 * embed_dim, 3 * embed_dim)
        self.value = nn.Linear(3 * embed_dim, 3 * embed_dim)
        self.output_linear = nn.Linear(self.embed_dim, embed_dim)  # New output laye
    def forward(self, x):
        q = self.query(x)
        k = self.key(x)
        v = self.value(x)
        attn_weights = torch.softmax(q @ k.transpose(-2, -1) / math.sqrt(self.embed_
        out = attn_weights @ v
        out = self.output_linear(out)  # Apply output layer
        return out
```

In [3]:
```
class Encoder(nn.Module):
    def __init__(self, input_dim, embed_dim, latent_dim):
        super(Encoder, self).__init__()
        self.cyclic_encoding = CyclicEncoding(embed_dim)
        self.self_attention_c = SelfAttention_C(embed_dim)
        self.linear1 = nn.Linear(input_dim, embed_dim)
        self.linear2 = nn.Linear(embed_dim, latent_dim)

    def forward(self, x):
        x = self.linear1(x)
        x = self.cyclic_encoding(x)
        x = self.self_attention_c(x)
        x = self.linear2(x)
        return x
```

In [4]:
```
class Decoder(nn.Module):
    def __init__(self, embed_dim, output_dim, latent_dim):
        super(Decoder, self).__init__()
        self.linear1 = nn.Linear(latent_dim, embed_dim)
        self.linear2 = nn.Linear(embed_dim, output_dim)

    def forward(self, x):
        x = self.linear1(x)
```

```
        x = self.linear2(x)
        return x
```

In [5]:

```
class Autoencoder(nn.Module):
    def __init__(self, input_dim, embed_dim, latent_dim, output_dim):
        super(Autoencoder, self).__init__()
        self.encoder = Encoder(input_dim, embed_dim, latent_dim)
        self.decoder = Decoder(embed_dim, output_dim, latent_dim)

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

In [6]:

```
class CyclicEncoding(nn.Module):
    def __init__(self, embed_dim):
        super(CyclicEncoding, self).__init__()
        self.embed_dim = embed_dim
    def forward(self, x):
        # Assuming x is of shape (batch_size, sequence_length, embed_dim)
        # And each value in x is a timestamp
        # We will encode it into a 2D cyclic encoding
        x1 = (x / (6*12)) % 1  # Convert timestamp to fraction of the day
        x1 = x1 * 2  * np.pi  # Convert to radians
        x1 = torch.cat([torch.sin(x1), torch.cos(x1)], dim=-1)  # Apply sine and cos
        x = torch.cat([x, x1], dim=-1)  # Concatenate original features with cyclica
        return x
```

In [7]:

```
# Load the data
df = pd.read_csv('jena_climate_2009_2016.csv')
df= df
# Display the first few rows of the dataframe
print(df.head())
```

```
             Date Time  p (mbar)  T (degC)  Tpot (K)  Tdew (degC)  rh (%)  \
0  01.01.2009 00:10:00    996.52     -8.02    265.40        -8.90    93.3
1  01.01.2009 00:20:00    996.57     -8.41    265.01        -9.28    93.4
2  01.01.2009 00:30:00    996.53     -8.51    264.91        -9.31    93.9
3  01.01.2009 00:40:00    996.51     -8.31    265.12        -9.07    94.2
4  01.01.2009 00:50:00    996.51     -8.27    265.15        -9.04    94.1

   VPmax (mbar)  VPact (mbar)  VPdef (mbar)  sh (g/kg)  H2OC (mmol/mol)  \
0          3.33          3.11          0.22       1.94             3.12
1          3.23          3.02          0.21       1.89             3.03
2          3.21          3.01          0.20       1.88             3.02
3          3.26          3.07          0.19       1.92             3.08
4          3.27          3.08          0.19       1.92             3.09

   rho (g/m**3)  wv (m/s)  max. wv (m/s)  wd (deg)
0       1307.75      1.03           1.75     152.3
1       1309.80      0.72           1.50     136.1
2       1310.24      0.19           0.63     171.6
3       1309.19      0.34           0.50     198.0
4       1309.00      0.32           0.63     214.3
```

In [8]:

```
# Convert the date-time column to datetime format
df['Date Time'] = pd.to_datetime(df['Date Time'], format='%d.%m.%Y %H:%M:%S')

# Convert datetime to unix timestamp
df['Date Time'] = (df['Date Time'] - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s
```

```
# Normalize the numerical columns
#for col in df.columns[1:]:
   # df[col] = (df[col] - df[col].mean()) / df[col].std()

# Handle missing values (this is just an example, you should choose a method that ma
df.fillna(df.mean(), inplace=True)

# Display the first few rows of the dataframe after preprocessing
print(df.head())
```

```
    Date Time  p (mbar)  T (degC)  Tpot (K)  Tdew (degC)  rh (%)  \
0  1230768600    996.52     -8.02    265.40        -8.90    93.3
1  1230769200    996.57     -8.41    265.01        -9.28    93.4
2  1230769800    996.53     -8.51    264.91        -9.31    93.9
3  1230770400    996.51     -8.31    265.12        -9.07    94.2
4  1230771000    996.51     -8.27    265.15        -9.04    94.1

   VPmax (mbar)  VPact (mbar)  VPdef (mbar)  sh (g/kg)  H2OC (mmol/mol)  \
0          3.33          3.11          0.22       1.94             3.12
1          3.23          3.02          0.21       1.89             3.03
2          3.21          3.01          0.20       1.88             3.02
3          3.26          3.07          0.19       1.92             3.08
4          3.27          3.08          0.19       1.92             3.09

   rho (g/m**3)  wv (m/s)  max. wv (m/s)  wd (deg)
0       1307.75      1.03           1.75     152.3
1       1309.80      0.72           1.50     136.1
2       1310.24      0.19           0.63     171.6
3       1309.19      0.34           0.50     198.0
4       1309.00      0.32           0.63     214.3
```

In [9]:

```
sequence_length = 6

# Create a list to hold the sequences
sequences = []

# Loop over the data to create sequences
for i in range(len(df) - sequence_length):
    # Get a sequence of length `sequence_length` starting at index `i`
    sequence = df.iloc[i:i+sequence_length].values
    # Add the sequence to the list
    sequences.append(sequence)

# Convert the list of sequences into a numpy array
sequences = np.array(sequences)

# Print the shape of the sequences array
print(sequences.shape)
```

```
(420545, 6, 15)
```

In [10]:

```
# Split the sequences into training and test sets
train_size = int(0.8 * len(sequences))
train_sequences = sequences[:train_size]
test_sequences = sequences[train_size:]

# Print the shapes of the training and test sets
print(train_sequences.shape)
print(test_sequences.shape)
```

```
(336436, 6, 15)
(84109, 6, 15)
```

In [11]:

```
# Split the training sequences into TR and VL sets
train_size = int(0.9 * len(test_sequences))
TR_sequences = test_sequences[:train_size]
VR_sequences = test_sequences[train_size:]

# Print the shapes of the training and test sets
print(TR_sequences.shape)
print(VR_sequences.shape)
```

```
(75698, 6, 15)
(8411, 6, 15)
```

In [12]:

```
# Define the device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define the model
model =Autoencoder(15, 4, 2, 15).to(device)

# Define the loss function and optimizer
criterion = nn.SmoothL1Loss()
optimizer = optim.Adam(model.parameters(), weight_decay=1e-5)  # Added weight decay
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.2,

# Convert the training sequences to a tensor
train_sequences_tensor = torch.tensor(TR_sequences).float().to(device)
val_sequences_tensor = torch.tensor(VR_sequences).float().to(device)

# Define the number of epochs
num_epochs = 1000

# Early stopping parameters
best_val_loss = float('inf')
patience = 50  # Number of epochs to wait after last time validation loss improved.
patience_counter = 0

# Train the model
for epoch in range(num_epochs):
    # Forward pass
    outputs = model(train_sequences_tensor)
    train_loss = criterion(outputs, train_sequences_tensor)

    # Backward pass and optimization
    optimizer.zero_grad()
    train_loss.backward()
    optimizer.step()



    # Validation phase
    model.eval()
    with torch.no_grad():
        val_outputs = model(val_sequences_tensor)
        val_loss = criterion(val_outputs, val_sequences_tensor)

    # Early stopping check
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        patience_counter = 0
    else:
        patience_counter += 1
        if patience_counter > patience:
            print(f'Early stopping triggered Best Training Loss: {best_val_loss.item
```

```
            break

        # Adjust learning rate
        scheduler.step(val_loss)

    # Print the loss for this epoch
    print(f'Epoch {epoch+1}/{num_epochs}, Training Loss: {train_loss.item()}, Va
```

```
Epoch 1/1000, Training Loss: 112378312.0, Validation Loss: 113582520.0
Epoch 2/1000, Training Loss: 111626712.0, Validation Loss: 112812680.0
Epoch 3/1000, Training Loss: 110870120.0, Validation Loss: 112071432.0
Epoch 4/1000, Training Loss: 110141648.0, Validation Loss: 111374328.0
Epoch 5/1000, Training Loss: 109456536.0, Validation Loss: 110698112.0
Epoch 6/1000, Training Loss: 108791976.0, Validation Loss: 110044400.0
Epoch 7/1000, Training Loss: 108149512.0, Validation Loss: 109414152.0
Epoch 8/1000, Training Loss: 107530112.0, Validation Loss: 108810568.0
Epoch 9/1000, Training Loss: 106936928.0, Validation Loss: 108232344.0
Epoch 10/1000, Training Loss: 106368664.0, Validation Loss: 107665512.0
Epoch 11/1000, Training Loss: 105811600.0, Validation Loss: 107118576.0
Epoch 12/1000, Training Loss: 105274072.0, Validation Loss: 106593112.0
Epoch 13/1000, Training Loss: 104757648.0, Validation Loss: 106084104.0
Epoch 14/1000, Training Loss: 104257424.0, Validation Loss: 105587008.0
Epoch 15/1000, Training Loss: 103768864.0, Validation Loss: 105096976.0
Epoch 16/1000, Training Loss: 103287280.0, Validation Loss: 104618984.0
Epoch 17/1000, Training Loss: 102817512.0, Validation Loss: 104151416.0
Epoch 18/1000, Training Loss: 102358008.0, Validation Loss: 103695400.0
Epoch 19/1000, Training Loss: 101909832.0, Validation Loss: 103249016.0
Epoch 20/1000, Training Loss: 101471136.0, Validation Loss: 102817720.0
Epoch 21/1000, Training Loss: 101047280.0, Validation Loss: 102396624.0
Epoch 22/1000, Training Loss: 100633432.0, Validation Loss: 101984200.0
Epoch 23/1000, Training Loss: 100228128.0, Validation Loss: 101579160.0
Epoch 24/1000, Training Loss: 99830032.0, Validation Loss: 101182640.0
Epoch 25/1000, Training Loss: 99440352.0, Validation Loss: 100792264.0
Epoch 26/1000, Training Loss: 99056696.0, Validation Loss: 100408528.0
Epoch 27/1000, Training Loss: 98679568.0, Validation Loss: 100031120.0
Epoch 28/1000, Training Loss: 98308648.0, Validation Loss: 99659864.0
Epoch 29/1000, Training Loss: 97943784.0, Validation Loss: 99292752.0
Epoch 30/1000, Training Loss: 97583016.0, Validation Loss: 98928792.0
Epoch 31/1000, Training Loss: 97225320.0, Validation Loss: 98831360.0
Epoch 32/1000, Training Loss: 97129568.0, Validation Loss: 99065240.0
Epoch 33/1000, Training Loss: 97359416.0, Validation Loss: 99249024.0
Epoch 34/1000, Training Loss: 97540040.0, Validation Loss: 99388712.0
Epoch 35/1000, Training Loss: 97677312.0, Validation Loss: 99490264.0
Epoch 36/1000, Training Loss: 97777128.0, Validation Loss: 99556576.0
Epoch 37/1000, Training Loss: 97842288.0, Validation Loss: 99591912.0
Epoch 38/1000, Training Loss: 97877008.0, Validation Loss: 99600064.0
Epoch 39/1000, Training Loss: 97885016.0, Validation Loss: 99583832.0
Epoch 40/1000, Training Loss: 97869072.0, Validation Loss: 99545720.0
Epoch 41/1000, Training Loss: 97831616.0, Validation Loss: 99489064.0
Epoch 42/1000, Training Loss: 97775936.0, Validation Loss: 99415784.0
Epoch 00012: reducing learning rate of group 0 to 2.0000e-04.
Epoch 43/1000, Training Loss: 97703920.0, Validation Loss: 99327984.0
Epoch 44/1000, Training Loss: 97617616.0, Validation Loss: 99307712.0
Epoch 45/1000, Training Loss: 97597720.0, Validation Loss: 99285144.0
Epoch 46/1000, Training Loss: 97575528.0, Validation Loss: 99260400.0
Epoch 47/1000, Training Loss: 97551208.0, Validation Loss: 99233600.0
Epoch 48/1000, Training Loss: 97524864.0, Validation Loss: 99205024.0
Epoch 49/1000, Training Loss: 97496792.0, Validation Loss: 99175456.0
Epoch 50/1000, Training Loss: 97467728.0, Validation Loss: 99150488.0
Epoch 51/1000, Training Loss: 97443200.0, Validation Loss: 99124392.0
Epoch 52/1000, Training Loss: 97417536.0, Validation Loss: 99097224.0
Epoch 53/1000, Training Loss: 97390840.0, Validation Loss: 99069176.0
Epoch 54/1000, Training Loss: 97363272.0, Validation Loss: 99040248.0
Epoch 55/1000, Training Loss: 97334840.0, Validation Loss: 99010560.0
```

```
Epoch 56/1000, Training Loss: 97305672.0, Validation Loss: 98980208.0
Epoch 57/1000, Training Loss: 97275848.0, Validation Loss: 98949264.0
Epoch 58/1000, Training Loss: 97245432.0, Validation Loss: 98917712.0
Epoch 59/1000, Training Loss: 97214424.0, Validation Loss: 98885904.0
Epoch 60/1000, Training Loss: 97183168.0, Validation Loss: 98856432.0
Epoch 61/1000, Training Loss: 97154200.0, Validation Loss: 98829680.0
Epoch 62/1000, Training Loss: 97127912.0, Validation Loss: 98804536.0
Epoch 63/1000, Training Loss: 97103200.0, Validation Loss: 98787528.0
Epoch 64/1000, Training Loss: 97086488.0, Validation Loss: 98768784.0
Epoch 65/1000, Training Loss: 97068056.0, Validation Loss: 98768288.0
Epoch 66/1000, Training Loss: 97067560.0, Validation Loss: 98782768.0
Epoch 67/1000, Training Loss: 97081784.0, Validation Loss: 98795184.0
Epoch 68/1000, Training Loss: 97093992.0, Validation Loss: 98804360.0
Epoch 69/1000, Training Loss: 97103032.0, Validation Loss: 98808912.0
Epoch 70/1000, Training Loss: 97107496.0, Validation Loss: 98808888.0
Epoch 71/1000, Training Loss: 97107464.0, Validation Loss: 98802312.0
Epoch 72/1000, Training Loss: 97101008.0, Validation Loss: 98792624.0
Epoch 73/1000, Training Loss: 97091480.0, Validation Loss: 98780312.0
Epoch 74/1000, Training Loss: 97079392.0, Validation Loss: 98766784.0
Epoch 75/1000, Training Loss: 97066088.0, Validation Loss: 98748032.0
Epoch 76/1000, Training Loss: 97047656.0, Validation Loss: 98743400.0
Epoch 77/1000, Training Loss: 97043104.0, Validation Loss: 98747064.0
Epoch 78/1000, Training Loss: 97046712.0, Validation Loss: 98750960.0
Epoch 79/1000, Training Loss: 97050552.0, Validation Loss: 98755480.0
Epoch 80/1000, Training Loss: 97054984.0, Validation Loss: 98757296.0
Epoch 81/1000, Training Loss: 97056760.0, Validation Loss: 98757808.0
Epoch 82/1000, Training Loss: 97057272.0, Validation Loss: 98757736.0
Epoch 83/1000, Training Loss: 97057200.0, Validation Loss: 98756056.0
Epoch 84/1000, Training Loss: 97055552.0, Validation Loss: 98751032.0
Epoch 85/1000, Training Loss: 97050608.0, Validation Loss: 98743048.0
Epoch 86/1000, Training Loss: 97042760.0, Validation Loss: 98732448.0
Epoch 87/1000, Training Loss: 97032336.0, Validation Loss: 98728416.0
Epoch 88/1000, Training Loss: 97028384.0, Validation Loss: 98729736.0
Epoch 89/1000, Training Loss: 97029688.0, Validation Loss: 98735312.0
Epoch 90/1000, Training Loss: 97035168.0, Validation Loss: 98738104.0
Epoch 91/1000, Training Loss: 97037904.0, Validation Loss: 98734712.0
Epoch 92/1000, Training Loss: 97034560.0, Validation Loss: 98733448.0
Epoch 93/1000, Training Loss: 97033320.0, Validation Loss: 98729248.0
Epoch 94/1000, Training Loss: 97029192.0, Validation Loss: 98723272.0
Epoch 95/1000, Training Loss: 97023344.0, Validation Loss: 98723400.0
Epoch 96/1000, Training Loss: 97023456.0, Validation Loss: 98726784.0
Epoch 97/1000, Training Loss: 97026784.0, Validation Loss: 98730784.0
Epoch 98/1000, Training Loss: 97030704.0, Validation Loss: 98729240.0
Epoch 99/1000, Training Loss: 97029184.0, Validation Loss: 98722568.0
Epoch 100/1000, Training Loss: 97022624.0, Validation Loss: 98718192.0
Epoch 101/1000, Training Loss: 97018344.0, Validation Loss: 98722432.0
Epoch 00057: reducing learning rate of group 0 to 4.0000e-05.
Epoch 102/1000, Training Loss: 97022504.0, Validation Loss: 98725400.0
Epoch 103/1000, Training Loss: 97025416.0, Validation Loss: 98725048.0
Epoch 104/1000, Training Loss: 97025080.0, Validation Loss: 98723936.0
Epoch 105/1000, Training Loss: 97023984.0, Validation Loss: 98722872.0
Epoch 106/1000, Training Loss: 97022944.0, Validation Loss: 98721056.0
Epoch 107/1000, Training Loss: 97021160.0, Validation Loss: 98718280.0
Epoch 108/1000, Training Loss: 97018408.0, Validation Loss: 98714656.0
Epoch 109/1000, Training Loss: 97014864.0, Validation Loss: 98716216.0
Epoch 110/1000, Training Loss: 97016392.0, Validation Loss: 98718552.0
Epoch 111/1000, Training Loss: 97018712.0, Validation Loss: 98720040.0
Epoch 112/1000, Training Loss: 97020152.0, Validation Loss: 98720744.0
Epoch 113/1000, Training Loss: 97020840.0, Validation Loss: 98720760.0
Epoch 114/1000, Training Loss: 97020872.0, Validation Loss: 98720168.0
Epoch 115/1000, Training Loss: 97020280.0, Validation Loss: 98719008.0
Epoch 116/1000, Training Loss: 97019152.0, Validation Loss: 98717360.0
Epoch 117/1000, Training Loss: 97017520.0, Validation Loss: 98716064.0
Epoch 118/1000, Training Loss: 97016232.0, Validation Loss: 98714920.0
```

```
Epoch 00073: reducing learning rate of group 0 to 8.0000e-06.
Epoch 119/1000, Training Loss: 97015120.0, Validation Loss: 98716136.0
Epoch 120/1000, Training Loss: 97016312.0, Validation Loss: 98716360.0
Epoch 121/1000, Training Loss: 97016544.0, Validation Loss: 98716472.0
Epoch 122/1000, Training Loss: 97016664.0, Validation Loss: 98716408.0
Epoch 123/1000, Training Loss: 97016584.0, Validation Loss: 98716160.0
Epoch 124/1000, Training Loss: 97016352.0, Validation Loss: 98715752.0
Epoch 125/1000, Training Loss: 97015944.0, Validation Loss: 98715264.0
Epoch 126/1000, Training Loss: 97015472.0, Validation Loss: 98714816.0
Epoch 127/1000, Training Loss: 97015008.0, Validation Loss: 98714232.0
Epoch 128/1000, Training Loss: 97014440.0, Validation Loss: 98713936.0
Epoch 129/1000, Training Loss: 97014136.0, Validation Loss: 98714376.0
Epoch 130/1000, Training Loss: 97014576.0, Validation Loss: 98714640.0
Epoch 131/1000, Training Loss: 97014856.0, Validation Loss: 98714784.0
Epoch 00084: reducing learning rate of group 0 to 1.6000e-06.
Epoch 132/1000, Training Loss: 97014984.0, Validation Loss: 98714808.0
Epoch 133/1000, Training Loss: 97015008.0, Validation Loss: 98714784.0
Epoch 134/1000, Training Loss: 97015000.0, Validation Loss: 98714776.0
Epoch 135/1000, Training Loss: 97014968.0, Validation Loss: 98714696.0
Epoch 136/1000, Training Loss: 97014912.0, Validation Loss: 98714640.0
Epoch 137/1000, Training Loss: 97014840.0, Validation Loss: 98714568.0
Epoch 138/1000, Training Loss: 97014776.0, Validation Loss: 98714472.0
Epoch 139/1000, Training Loss: 97014688.0, Validation Loss: 98714376.0
Epoch 140/1000, Training Loss: 97014576.0, Validation Loss: 98714264.0
Epoch 141/1000, Training Loss: 97014480.0, Validation Loss: 98714144.0
Epoch 142/1000, Training Loss: 97014360.0, Validation Loss: 98714008.0
Epoch 143/1000, Training Loss: 97014216.0, Validation Loss: 98713880.0
Epoch 144/1000, Training Loss: 97014096.0, Validation Loss: 98713792.0
Epoch 00095: reducing learning rate of group 0 to 3.2000e-07.
Epoch 145/1000, Training Loss: 97014008.0, Validation Loss: 98713864.0
Epoch 146/1000, Training Loss: 97014088.0, Validation Loss: 98713896.0
Epoch 147/1000, Training Loss: 97014112.0, Validation Loss: 98713912.0
Epoch 148/1000, Training Loss: 97014128.0, Validation Loss: 98713936.0
Epoch 149/1000, Training Loss: 97014144.0, Validation Loss: 98713936.0
Epoch 150/1000, Training Loss: 97014144.0, Validation Loss: 98713912.0
Epoch 151/1000, Training Loss: 97014144.0, Validation Loss: 98713912.0
Epoch 152/1000, Training Loss: 97014128.0, Validation Loss: 98713888.0
Epoch 153/1000, Training Loss: 97014112.0, Validation Loss: 98713888.0
Epoch 154/1000, Training Loss: 97014104.0, Validation Loss: 98713880.0
Epoch 155/1000, Training Loss: 97014088.0, Validation Loss: 98713856.0
Epoch 00106: reducing learning rate of group 0 to 6.4000e-08.
Epoch 156/1000, Training Loss: 97014072.0, Validation Loss: 98713848.0
Epoch 157/1000, Training Loss: 97014056.0, Validation Loss: 98713832.0
Epoch 158/1000, Training Loss: 97014048.0, Validation Loss: 98713808.0
Epoch 159/1000, Training Loss: 97014048.0, Validation Loss: 98713808.0
Epoch 160/1000, Training Loss: 97014032.0, Validation Loss: 98713800.0
Epoch 161/1000, Training Loss: 97014024.0, Validation Loss: 98713792.0
Epoch 162/1000, Training Loss: 97014024.0, Validation Loss: 98713792.0
Epoch 163/1000, Training Loss: 97014008.0, Validation Loss: 98713768.0
Epoch 164/1000, Training Loss: 97014000.0, Validation Loss: 98713768.0
Epoch 165/1000, Training Loss: 97013984.0, Validation Loss: 98713768.0
Epoch 166/1000, Training Loss: 97013984.0, Validation Loss: 98713768.0
Epoch 167/1000, Training Loss: 97013984.0, Validation Loss: 98713768.0
Epoch 00117: reducing learning rate of group 0 to 1.2800e-08.
Epoch 168/1000, Training Loss: 97013984.0, Validation Loss: 98713768.0
Epoch 169/1000, Training Loss: 97013976.0, Validation Loss: 98713768.0
Epoch 170/1000, Training Loss: 97013976.0, Validation Loss: 98713768.0
Epoch 171/1000, Training Loss: 97013976.0, Validation Loss: 98713768.0
Epoch 172/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 173/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 174/1000, Training Loss: 97013976.0, Validation Loss: 98713768.0
Epoch 175/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 176/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 177/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
```

```
Epoch 178/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 179/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 00128: reducing learning rate of group 0 to 2.5600e-09.
Epoch 180/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 181/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 182/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 183/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 184/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 185/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 186/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 187/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 188/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 189/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 190/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 191/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 192/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 193/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 194/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 195/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 196/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 197/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 198/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 199/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 200/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 201/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 202/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 203/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 204/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 205/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 206/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 207/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 208/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 209/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 210/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 211/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 212/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 213/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 214/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 215/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 216/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 217/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 218/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 219/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 220/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 221/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Epoch 222/1000, Training Loss: 97013976.0, Validation Loss: 98713752.0
Early stopping triggered Best Training Loss: 98713752.0
```

In [13]:

```
print(outputs.shape)
encoded_features = outputs
```

```
torch.Size([75698, 6, 15])
```

In [14]:

```
# Split the test_sequences into training and test sets
train_size = int(0.8 * len(test_sequences))
TRtest_sequences = test_sequences[:train_size]
VRtest_sequences = test_sequences[train_size:]

# Print the shapes of the training and test sets
print(TRtest_sequences.shape)
print(VRtest_sequences.shape)
```

```
(67287, 6, 15)
(16822, 6, 15)
```

In [15]:

```
# Flatten the 6x15 matrices into 1D vectors (90 elements each)
flattened_features = encoded_features.view(encoded_features.shape[0], -1).detach().n
# Create a synthetic target, for example, summing up all elements of each vector
synthetic_target = np.sum(flattened_features, axis=1)
```

In [16]:

```
# Step 2: Split the data and train a regression model
X_train, X_test, y_train, y_test = train_test_split(flattened_features, synthetic_ta
regressor = LinearRegression()
regressor.fit(X_train, y_train)

# Step 3: Evaluate the model
y_pred = regressor.predict(X_test)
# Calculate metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))  # Or directly use sqrt from math
#msle = mean_squared_log_error(y_test, y_pred)
medae = median_absolute_error(y_test, y_pred)

# Print out the metrics
print(f'Below is linearRegression')
print(f'Mean Squared Error: {mse}')
print(f'Mean Absolute Error: {mae}')
print(f'Root Mean Squared Error: {rmse}')
#print(f'Mean Squared Logarithmic Error: {msle}')
print(f'Median Absolute Error: {medae}')

# Let's assume a polynomial degree of 2 for demonstration
poly_transformer = PolynomialFeatures(degree=2)

model_orig = Pipeline([
    ('poly', PolynomialFeatures(degree=2)),
    ('linear', LinearRegression())
])
model_orig.fit(X_train, y_train)
y_pred_orig = model_orig.predict(X_test)
# Calculate metrics
mse_orig = mean_squared_error(y_test, y_pred_orig)
mae_orig = mean_absolute_error(y_test, y_pred_orig)
rmse_orig = np.sqrt(mean_squared_error(y_test, y_pred_orig))  # Or directly use sqrt
#msle_orig = mean_squared_log_error(y_test, y_pred_orig)
medae_orig = median_absolute_error(y_test, y_pred_orig)
# Print out the metrics
print(f'Below is Polunomial linearRegression')
print(f'Mean Squared Error on Polynomial: {mse_orig}')
print(f'Mean Absolute Error: {mae_orig}')
print(f'Root Mean Squared Error: {rmse_orig}')
#print(f'Mean Squared Logarithmic Error: {msle_orig}')
print(f'Median Absolute Error: {medae_orig}')

# Add Support Vector Regression (SVR)
svr = SVR()
svr.fit(X_train, y_train)
y_pred_svr = svr.predict(X_test)
# Calculate metrics for SVR
mse_svr = mean_squared_error(y_test, y_pred_svr)
mae_svr = mean_absolute_error(y_test, y_pred_svr)
rmse_svr = np.sqrt(mean_squared_error(y_test, y_pred_svr))
#msle_svr = mean_squared_log_error(y_test, y_pred_svr)
```

```
medae_svr = median_absolute_error(y_test, y_pred_svr)
# Print out the metrics for SVR
print(f'Below is Support Vector Regression')
print(f'Mean Squared Error: {mse_svr}')
print(f'Mean Absolute Error: {mae_svr}')
print(f'Root Mean Squared Error: {rmse_svr}')
#print(f'Mean Squared Logarithmic Error: {msle_svr}')
print(f'Median Absolute Error: {medae_svr}')


# Add K-Nearest Neighbors Regression
knn = KNeighborsRegressor(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
# Calculate metrics for KNN
mse_knn = mean_squared_error(y_test, y_pred_knn)
mae_knn = mean_absolute_error(y_test, y_pred_knn)
rmse_knn = np.sqrt(mean_squared_error(y_test, y_pred_knn))
#msle_knn = mean_squared_log_error(y_test, y_pred_knn)
medae_knn = median_absolute_error(y_test, y_pred_knn)
# Print out the metrics for KNN
print(f'Below is K-Nearest Neighbors Regression')
print(f'Mean Squared Error: {mse_knn}')
print(f'Mean Absolute Error: {mae_knn}')
print(f'Root Mean Squared Error: {rmse_knn}')
#print(f'Mean Squared Logarithmic Error: {msle_knn}')
print(f'Median Absolute Error: {medae_knn}')
```

```
Below is linearRegression
Mean Squared Error: 9.305549042437633e-08
Mean Absolute Error: 0.00024150409444700927
Root Mean Squared Error: 0.000305049994494766
Median Absolute Error: 0.000244140625
Below is Polunomial linearRegression
Mean Squared Error on Polynomial: 4.574736067297636e-06
Mean Absolute Error: 0.001599870971404016
Root Mean Squared Error: 0.002138863317668438
Median Absolute Error: 0.0013427734375
Below is Support Vector Regression
Mean Squared Error: 0.10717039026513413
Mean Absolute Error: 0.1068060156904755
Root Mean Squared Error: 0.32736889019137744
Median Absolute Error: 0.07199441225498049
Below is K-Nearest Neighbors Regression
Mean Squared Error: 0.11792375892400742
Mean Absolute Error: 0.026895927265286446
Root Mean Squared Error: 0.3434002995491028
Median Absolute Error: 0.0
```

A.3 Cyler layer 12 with full data

In [1]:
```
import torch
from torch import nn
import torch.nn as nn
import torch.optim as optim
import pandas as pd
import numpy as np
import math
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score, mean
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
```

In [2]:
```
class SelfAttention_C(nn.Module):

    def __init__(self, embed_dim):
        super(SelfAttention_C, self).__init__()
        self.embed_dim = 3 *embed_dim
        self.query = nn.Linear(3 * embed_dim, 3 * embed_dim)
        self.key = nn.Linear(3 * embed_dim, 3 * embed_dim)
        self.value = nn.Linear(3 * embed_dim, 3 * embed_dim)
        self.output_linear = nn.Linear(self.embed_dim, embed_dim)  # New output layer
    def forward(self, x):
        q = self.query(x)
        k = self.key(x)
        v = self.value(x)
        attn_weights = torch.softmax(q @ k.transpose(-2, -1) / math.sqrt(self.embed_
        out = attn_weights @ v
        out = self.output_linear(out)  # Apply output layer
        return out
```

In [3]:
```
class Encoder(nn.Module):
    def __init__(self, input_dim, embed_dim, latent_dim):
        super(Encoder, self).__init__()
        self.cyclic_encoding = CyclicEncoding(embed_dim)
        self.self_attention_c = SelfAttention_C(embed_dim)
        self.linear1 = nn.Linear(input_dim, embed_dim)
        self.linear2 = nn.Linear(embed_dim, latent_dim)

    def forward(self, x):
        x = self.linear1(x)
        x = self.cyclic_encoding(x)
        x = self.self_attention_c(x)
        x = self.linear2(x)
        return x
```

In [4]:
```
class Decoder(nn.Module):
    def __init__(self, embed_dim, output_dim, latent_dim):
        super(Decoder, self).__init__()
        self.linear1 = nn.Linear(latent_dim, embed_dim)
        self.linear2 = nn.Linear(embed_dim, output_dim)

    def forward(self, x):
        x = self.linear1(x)
        x = self.linear2(x)
        return x
```

In [5]:
```
class Autoencoder(nn.Module):
    def __init__(self, input_dim, embed_dim, latent_dim, output_dim):
        super(Autoencoder, self).__init__()
        self.encoder = Encoder(input_dim, embed_dim, latent_dim)
        self.decoder = Decoder(embed_dim, output_dim, latent_dim)

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

In [6]:
```
class CyclicEncoding(nn.Module):
    def __init__(self, embed_dim):
        super(CyclicEncoding, self).__init__()
        self.embed_dim = embed_dim
    def forward(self, x):
        # Assuming x is of shape (batch_size, sequence_length, embed_dim)
        # And each value in x is a timestamp
        # We will encode it into a 2D cyclic encoding
        x1 = (x / (6*24)) % 1  # Convert timestamp to fraction of the day
        x1 = x1 * 2  * np.pi  # Convert to radians
        x1 = torch.cat([torch.sin(x1), torch.cos(x1)], dim=-1)  # Apply sine and cos
        x = torch.cat([x, x1], dim=-1)  # Concatenate original features with cyclica
        return x
```

In [7]:
```
# Load the data
df = pd.read_csv('jena_climate_2009_2016.csv')
df= df
# Display the first few rows of the dataframe
print(df.head())
```

```
             Date Time  p (mbar)  T (degC)  Tpot (K)  Tdew (degC)  rh (%)  \
0  01.01.2009 00:10:00    996.52     -8.02    265.40        -8.90    93.3
1  01.01.2009 00:20:00    996.57     -8.41    265.01        -9.28    93.4
2  01.01.2009 00:30:00    996.53     -8.51    264.91        -9.31    93.9
3  01.01.2009 00:40:00    996.51     -8.31    265.12        -9.07    94.2
4  01.01.2009 00:50:00    996.51     -8.27    265.15        -9.04    94.1

   VPmax (mbar)  VPact (mbar)  VPdef (mbar)  sh (g/kg)  H2OC (mmol/mol)  \
0          3.33          3.11          0.22       1.94             3.12
1          3.23          3.02          0.21       1.89             3.03
2          3.21          3.01          0.20       1.88             3.02
3          3.26          3.07          0.19       1.92             3.08
4          3.27          3.08          0.19       1.92             3.09

   rho (g/m**3)  wv (m/s)  max. wv (m/s)  wd (deg)
0       1307.75      1.03           1.75     152.3
1       1309.80      0.72           1.50     136.1
2       1310.24      0.19           0.63     171.6
3       1309.19      0.34           0.50     198.0
4       1309.00      0.32           0.63     214.3
```

In [8]:
```
# Convert the date-time column to datetime format
df['Date Time'] = pd.to_datetime(df['Date Time'], format='%d.%m.%Y %H:%M:%S')

# Convert datetime to unix timestamp
df['Date Time'] = (df['Date Time'] - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s

# Normalize the numerical columns
#for col in df.columns[1:]:
   # df[col] = (df[col] - df[col].mean()) / df[col].std()

# Handle missing values (this is just an example, you should choose a method that mak
df.fillna(df.mean(), inplace=True)

# Display the first few rows of the dataframe after preprocessing
print(df.head())
```

```
    Date Time  p (mbar)  T (degC)  Tpot (K)  Tdew (degC)  rh (%)  \
0  1230768600    996.52     -8.02    265.40        -8.90    93.3
1  1230769200    996.57     -8.41    265.01        -9.28    93.4
2  1230769800    996.53     -8.51    264.91        -9.31    93.9
3  1230770400    996.51     -8.31    265.12        -9.07    94.2
4  1230771000    996.51     -8.27    265.15        -9.04    94.1

   VPmax (mbar)  VPact (mbar)  VPdef (mbar)  sh (g/kg)  H2OC (mmol/mol)  \
0          3.33          3.11          0.22       1.94             3.12
1          3.23          3.02          0.21       1.89             3.03
2          3.21          3.01          0.20       1.88             3.02
3          3.26          3.07          0.19       1.92             3.08
4          3.27          3.08          0.19       1.92             3.09

   rho (g/m**3)  wv (m/s)  max. wv (m/s)  wd (deg)
0       1307.75      1.03           1.75     152.3
1       1309.80      0.72           1.50     136.1
2       1310.24      0.19           0.63     171.6
3       1309.19      0.34           0.50     198.0
4       1309.00      0.32           0.63     214.3
```

In [9]:
```
sequence_length = 6

# Create a list to hold the sequences
sequences = []

# Loop over the data to create sequences
for i in range(len(df) - sequence_length):
    # Get a sequence of length `sequence_length` starting at index `i`
    sequence = df.iloc[i:i+sequence_length].values
    # Add the sequence to the list
    sequences.append(sequence)

# Convert the list of sequences into a numpy array
sequences = np.array(sequences)

# Print the shape of the sequences array
print(sequences.shape)
```

```
(420545, 6, 15)
```

In [10]:
```
# Split the sequences into training and test sets
train_size = int(0.8 * len(sequences))
train_sequences = sequences[:train_size]
test_sequences = sequences[train_size:]

# Print the shapes of the training and test sets
print(train_sequences.shape)
print(test_sequences.shape)
```

```
(336436, 6, 15)
(84109, 6, 15)
```

In [11]:
```
# Split the training sequences into TR and VL sets
train_size = int(0.9 * len(test_sequences))
TR_sequences = test_sequences[:train_size]
VR_sequences = test_sequences[train_size:]

# Print the shapes of the training and test sets
print(TR_sequences.shape)
print(VR_sequences.shape)
```

```
(75698, 6, 15)
(8411, 6, 15)
```

In [12]:
```
# Define the device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define the model
model =Autoencoder(15, 4, 2, 15).to(device)

# Define the loss function and optimizer
criterion = nn.SmoothL1Loss()
optimizer = optim.Adam(model.parameters(), weight_decay=1e-5)  # Added weight decay
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.2,

# Convert the training sequences to a tensor
train_sequences_tensor = torch.tensor(TR_sequences).float().to(device)
val_sequences_tensor = torch.tensor(VR_sequences).float().to(device)

# Define the number of epochs
num_epochs = 1000

# Early stopping parameters
best_val_loss = float('inf')
patience = 50  # Number of epochs to wait after last time validation loss improved.
patience_counter = 0

# Train the model
for epoch in range(num_epochs):
    # Forward pass
    outputs = model(train_sequences_tensor)
    train_loss = criterion(outputs, train_sequences_tensor)

    # Backward pass and optimization
    optimizer.zero_grad()
    train_loss.backward()
    optimizer.step()



    # Validation phase
    model.eval()
    with torch.no_grad():
        val_outputs = model(val_sequences_tensor)
        val_loss = criterion(val_outputs, val_sequences_tensor)

    # Early stopping check
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        patience_counter = 0
    else:
        patience_counter += 1
        if patience_counter > patience:
            print(f'Early stopping triggered Best Training Loss: {best_val_loss.item(
            break

     # Adjust learning rate
        scheduler.step(val_loss)

    # Print the loss for this epoch
    print(f'Epoch {epoch+1}/{num_epochs}, Training Loss: {train_loss.item()}, Validat
```

```
Epoch 1/1000, Training Loss: 98692008.0, Validation Loss: 100287680.0
Epoch 2/1000, Training Loss: 98560792.0, Validation Loss: 100159152.0
Epoch 3/1000, Training Loss: 98434488.0, Validation Loss: 100029320.0
Epoch 4/1000, Training Loss: 98306880.0, Validation Loss: 99903272.0
Epoch 5/1000, Training Loss: 98183008.0, Validation Loss: 99780272.0
Epoch 6/1000, Training Loss: 98062136.0, Validation Loss: 99661128.0
Epoch 7/1000, Training Loss: 97945024.0, Validation Loss: 99545312.0
Epoch 8/1000, Training Loss: 97831216.0, Validation Loss: 99433416.0
Epoch 9/1000, Training Loss: 97721240.0, Validation Loss: 99324960.0
Epoch 10/1000, Training Loss: 97614664.0, Validation Loss: 99219528.0
Epoch 11/1000, Training Loss: 97511040.0, Validation Loss: 99118064.0
Epoch 12/1000, Training Loss: 97411328.0, Validation Loss: 99020160.0
Epoch 13/1000, Training Loss: 97315096.0, Validation Loss: 98924792.0
Epoch 14/1000, Training Loss: 97221384.0, Validation Loss: 98835496.0
Epoch 15/1000, Training Loss: 97133608.0, Validation Loss: 98747576.0
Epoch 16/1000, Training Loss: 97047200.0, Validation Loss: 98818088.0
Epoch 17/1000, Training Loss: 97116512.0, Validation Loss: 98899584.0
Epoch 18/1000, Training Loss: 97196600.0, Validation Loss: 98941760.0
Epoch 19/1000, Training Loss: 97238040.0, Validation Loss: 98962992.0
```

In [13]:
```
print(outputs.shape)
encoded_features = outputs
```

```
torch.Size([75698, 6, 15])
```

In [14]:
```
# Split the test_sequences into training and test sets
train_size = int(0.8 * len(test_sequences))
TRtest_sequences = test_sequences[:train_size]
VRtest_sequences = test_sequences[train_size:]

# Print the shapes of the training and test sets
print(TRtest_sequences.shape)
print(VRtest_sequences.shape)
```

```
(67287, 6, 15)
(16822, 6, 15)
```

In [15]:
```
# Flatten the 6x15 matrices into 1D vectors (90 elements each)
flattened_features = encoded_features.view(encoded_features.shape[0], -1).detach().n
# Create a synthetic target, for example, summing up all elements of each vector
synthetic_target = np.sum(flattened_features, axis=1)
```

In [16]:
```
# Step 2: Split the data and train a regression model
X_train, X_test, y_train, y_test = train_test_split(flattened_features, synthetic_ta
regressor = LinearRegression()
regressor.fit(X_train, y_train)

# Step 3: Evaluate the model
y_pred = regressor.predict(X_test)
# Calculate metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))  # Or directly use sqrt from math
#msle = mean_squared_log_error(y_test, y_pred)
medae = median_absolute_error(y_test, y_pred)

# Print out the metrics
print(f'Below is linearRegression')
print(f'Mean Squared Error: {mse}')
print(f'Mean Absolute Error: {mae}')
print(f'Root Mean Squared Error: {rmse}')
#print(f'Mean Squared Logarithmic Error: {msle}')
print(f'Median Absolute Error: {medae}')

# Let's assume a polynomial degree of 2 for demonstration
poly_transformer = PolynomialFeatures(degree=2)

model_orig = Pipeline([
    ('poly', PolynomialFeatures(degree=2)),
    ('linear', LinearRegression())
])
model_orig.fit(X_train, y_train)
y_pred_orig = model_orig.predict(X_test)
# Calculate metrics
mse_orig = mean_squared_error(y_test, y_pred_orig)
mae_orig = mean_absolute_error(y_test, y_pred_orig)
rmse_orig = np.sqrt(mean_squared_error(y_test, y_pred_orig))  # Or directly use sqrt
#msle_orig = mean_squared_log_error(y_test, y_pred_orig)
medae_orig = median_absolute_error(y_test, y_pred_orig)
# Print out the metrics
print(f'Below is Polunomial linearRegression')
print(f'Mean Squared Error on Polynomial: {mse_orig}')
print(f'Mean Absolute Error: {mae_orig}')
print(f'Root Mean Squared Error: {rmse_orig}')
#print(f'Mean Squared Logarithmic Error: {msle_orig}')
print(f'Median Absolute Error: {medae_orig}')

# Add Support Vector Regression (SVR)
svr = SVR()
svr.fit(X_train, y_train)
y_pred_svr = svr.predict(X_test)
# Calculate metrics for SVR
mse_svr = mean_squared_error(y_test, y_pred_svr)
mae_svr = mean_absolute_error(y_test, y_pred_svr)
rmse_svr = np.sqrt(mean_squared_error(y_test, y_pred_svr))
#msle_svr = mean_squared_log_error(y_test, y_pred_svr)
medae_svr = median_absolute_error(y_test, y_pred_svr)
# Print out the metrics for SVR
print(f'Below is Support Vector Regression')
print(f'Mean Squared Error: {mse_svr}')
print(f'Mean Absolute Error: {mae_svr}')
print(f'Root Mean Squared Error: {rmse_svr}')
#print(f'Mean Squared Logarithmic Error: {msle_svr}')
```

```
print(f'Median Absolute Error: {medae_svr}')


# Add K-Nearest Neighbors Regression
knn = KNeighborsRegressor(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
# Calculate metrics for KNN
mse_knn = mean_squared_error(y_test, y_pred_knn)
mae_knn = mean_absolute_error(y_test, y_pred_knn)
rmse_knn = np.sqrt(mean_squared_error(y_test, y_pred_knn))
#msle_knn = mean_squared_log_error(y_test, y_pred_knn)
medae_knn = median_absolute_error(y_test, y_pred_knn)
# Print out the metrics for KNN
print(f'Below is K-Nearest Neighbors Regression')
print(f'Mean Squared Error: {mse_knn}')
print(f'Mean Absolute Error: {mae_knn}')
print(f'Root Mean Squared Error: {rmse_knn}')
#print(f'Mean Squared Logarithmic Error: {msle_knn}')
print(f'Median Absolute Error: {medae_knn}')
```

```
Below is linearRegression
Mean Squared Error: 3.9787888539422056e-08
Mean Absolute Error: 0.00014874408952891827
Root Mean Squared Error: 0.00019946901011280715
Median Absolute Error: 0.0001277923583984375
Below is Polunomial linearRegression
Mean Squared Error on Polynomial: 2.0274090275052004e-05
Mean Absolute Error: 0.0021371108014136553
Root Mean Squared Error: 0.004502675961703062
Median Absolute Error: 0.0015277862548828125
Below is Support Vector Regression
Mean Squared Error: 34.00580940565318
Mean Absolute Error: 0.16093211726353499
Root Mean Squared Error: 5.83145002599295
Median Absolute Error: 0.054922733511372
Below is K-Nearest Neighbors Regression
Mean Squared Error: 0.01507758442312479
Mean Absolute Error: 0.007724801078438759
Root Mean Squared Error: 0.12279081344604492
Median Absolute Error: 0.0
```