MDPI

Article

Deconstructing transfomer-based time series forecasting

Filippo Garagnani [1] **and Vittorio Maniezzo** [2,*]

[1] University of Modena and Reggio Emilia; 298707@studenti.unimore.it
[2] Department of Computer Science, University of Bologna; vittorio.maniezzo@unibo.it
* Correspondence: vittorio.maniezzo@unibo.it;

Abstract: This paper provides a thorough breakdown of a fundamental Transformer-based architecture for forecasting univariate time series. We describe each processing step in detail, from input embedding and positional encoding to self-attention mechanisms and output projection, all of which are tailored specifically to sequential temporal data. By isolating and analyzing the role of each component, we demonstrate how transformers capture long-term dependencies in time series. We implement a simplified, interpretable transformer model and showcase it on a simple use case. We then validate it on a significant benchmark suite, including datasets from forecasting competitions and real-world applications. The aim of this work is to serve as both a practical guide and a foundation for future innovations in transformer-based forecasting.

Keywords: Data series forecasting; Transformer models; Machine learning

1. Introduction

Time series forecasting is a key topic in data science and has a wide range of applications in areas such as economics, energy, healthcare, logistics and environmental monitoring. Accurate forecasting enables informed decision-making, efficient resource allocation and risk mitigation. Traditional statistical methods such as ARIMA and exponential smoothing have long been used for this purpose and often produce good results when the necessary assumptions are met. However, these methods often struggle to capture complex patterns and long-term dependencies in the data, an area in which deep learning methods, especially sequence models such as recurrent neural networks (RNNs), have often demonstrated superior performance.

Transformer models, which were initially designed for natural language processing tasks, have recently shown remarkable success in understanding and generating sequential data, demonstrating their effectiveness in modelling long-range dependencies. Transformers are attention-based models that eliminate the need for recurrence and convolution, relying instead on a self-attention mechanism to capture global context across sequences. This architecture has had a revolutionary impact on NLP, and its application to time series forecasting is a rapidly growing area of research. Unlike natural language, time series are composed of continuous values and frequently exhibit seasonality, trends and influence from exogenous variables. While recent studies have proposed various adaptations of the transformer to address the unique properties of time series data, many of these models are treated as opaque black boxes. This means that there is limited clear understanding of how each component relates to temporal information and contributes to forecasting performance. Furthermore, the implementation details are frequently insufficiently specified, which makes reproducibility and interpretability challenging.

In this paper, our aim is to 'deconstruct' the forecasting process of transformer-based models when applied to univariate time series. Our objective is to provide a transparent,

Received:
Revised:
Accepted:
Published:

Citation: Garagnani, F.; Maniezzo, V. Deconstructing transfomer-based time series forecasting. *Algorithms* **2025**, *1*, 0. https://doi.org/

modular and pedagogical account of how univariate time series are transformed and processed within these models to generate future trajectories. We describe each processing step in detail, including data normalization, input windowing, positional encoding, encoder and decoder architecture, training strategies, and prediction generation. We highlight both the theoretical basis and the practical design choices, showing how each element contributes to predictive accuracy and attempting to bridge the gap between theoretical advancements and practical implementation.

To validate our methodology, we implemented and evaluated a basic transformer-based model on a significant benchmark of univariate time series. These were selected from the well-known M forecasting competition benchmarks and included macro- and microeconomic, industrial, and demographic series with varying statistical properties. The work combines theoretical explanation with empirical validation to serve as a comprehensive guide to deconstructing transformer-based time series forecasting, providing a tutorial for newcomers and a reference for practitioners.

The paper is structured as follows. Section 2 reviews the related works in time series forecasting, with particular emphasis on recent advancements leveraging Transformer architectures. Section 3 presents the workflow of a minimalist Transformer module to produce a multipoint forecast for a univariate time series. Section 4 analyses some results in applying the described architecture on benchmark data series and comparing it with another standard methodology.

2. Related work

Transformer modules for processing graph-like structures were originally introduced in [1] for a check-reading task, however it was only with [2] that the current transformer architecture for sequence modelling was proposed. Already the very first line of the introduction of this seminal paper frames the transformer architecture among established time series modelling and forecasting algorithms, citing LSTM [3] and GRU [4], i.e., the two main architectures for recurrent neural networks (RNN) applied to time series modelling.

Indeed, RNNs can be effective in capturing long-term dependencies among data, but suffer from slow training, vanishing gradients and high sensitivity to hyperparameter settings. Furthermore, the most well-established statistical methods, such as autoregressive integrated moving average (ARIMA) [5] and exponential smoothing (ETS) [6,7], are highly effective for stationary data but struggle with complex nonlinear patterns.

Transformer architectures are immune from these drawbacks because they make no stationarity assumption and capture global dependencies in input sequences without relying on recurrence, instead leveraging self-attention mechanisms enabling direct modeling of long-range dependencies. This supports full parallelization during training and allows the model to focus on relevant time steps, regardless of their position in the sequence.

A key challenge in applying transformers to univariate time series is the absence of the rich semantic tokens found in language. Unlike words in a sentence, time series values are continuous and lack discrete meaning, making it more difficult to learn meaningful attention patterns. Therefore, transformers need to be adapted for time series forecasting. Early adaptations of transformers for use with time series data include

Informer [8], which introduces a sparse self-attention mechanism to reduce the quadratic complexity of self-attention, making it more efficient for long sequence time series forecasting.

Autoformer [9], which integrates a series decomposition block within the transformer to separate trend and seasonal components obtaining a decomposition-based attention for better seasonal-trend modeling, improving interpretability and long-horizon forecasting accuracy.

FEDformer [10], which incorporates Fourier transformations into the attention mechanism to better exploit frequency-domain characteristics.

PatchTST [11], which employs a splitting mechanism to time series, converting multiple timesteps (i.e., a patch) into a single token, enabling a vanilla Transformer architecture to capture long-term dependencies with fewer tokens.

More recently, an explosion of adaptation has appeared, together with dedicated surveys and tutorials [12–14]. However, these models are often complex, incorporating multiple components such as static covariates, gating mechanisms, and probabilistic outputs. While these additions have shown empirical gains, they frequently obscure the core dynamics of the attention mechanism and make the models less accessible for analysis or adaptation.

By contrast, there is interest in minimal Transformer implementations, which isolate the effectiveness of self-attention in forecasting tasks. Informer has already demonstrated that a simplified Transformer, devoid of specialized components, can outperform classical models and RNNs on certain benchmarks when properly tuned. Similarly, [15] proposed the 'Temporal Attention' module within a lightweight Transformer framework, demonstrating that attention over timesteps alone can achieve comparable performance on univariate series. Other studies, such as iTransformer [16], investigate how the performance of standard Transformers is affected by different initialization, normalization, or positional encoding strategies. This sheds light on the contribution of each component to forecasting accuracy.

This body of work highlights the growing interest in demystifying transformer-based time series models by removing layers of abstraction. In line with this perspective, we propose a minimalist transformer specifically designed for univariate forecasting. We implement and evaluate a minimal version of the transformer encoder by maintaining only the essential self-attention and feedforward components. By simplifying the model, our aim is to understand the fundamental processes that enable transformers to succeed in this domain and to provide a transparent and reproducible basis for further research.

3. The transformer processing pipeline

This section describes the successive processing phases carried out by a minimalist transformer module to produce a multipoint forecast of a univariate data series. The relevant python code can be found in [17]. Successive processing stages are showcased on a tiny time series obtained by querying Google Trends [18] on the term "restaurant". The series shows a trend (at the time of writing, it is holiday season in Italy and interest in dining out is increasing) as well as seasonality, due to increased interest at weekends. We consider only 35 data points, 7 of which are kept for validation and 28 are used for training.

The series $s(t)$ is the following:

$$s(t) = (44, 48, 51, 48, 50, 63, 66, 48, 53, 56, 52, 57, 70, 67, 56, 60, 62, 60, 58, 75, 73, 59, 61, 65, \\ 63, 63, 78, 80, 63, 64, 67, 65, 70, 87, 84)$$

The data processing cycle consists of three main phases: encoding, decoding, and learning. The following subsections provide details on each phase, the reader is referred to [2] for a more exhaustive exposition.

3.1. Encoding

The objective of the encoding component (i.e., the *encoder*) in a Transformer architecture is to process an input sequence and convert the raw input into a contextually enriched

representation that captures both local and global dependencies. This representation can then be used effectively by the rest of the model.

The general structure of the encoding module is represented by the pseudocode in Algorithm **??**. All intermediate numerical results obtained for the restaurant series case study are reported in appendix A to enable the interested reader to immediately grasp the dimensionalities and values involved in this tiny example, and possibly to follow the corresponding evolution of a run of the code provided in [17]. In the case of the running example we used the following argument values: n=7, m=4, k=2, d_k=d_v=2, p=16.

The individual steps are detailed below.

Algorithm 1: The encoding module

procedure encoding(s,n,m,k,d_k,d_v,p) lgo:encoding
Input : A time series $\mathbf{s} = [s_i], i = 0, \ldots, t$
Output: A 2D matrix Z
`// Input Projection`
`// transform the array s into the 2D embedding matrix (tensor) X`
for $i = 0, \ldots, t$ **do**
  nc:seca1 Expand value s_i into a row $X_i = s_i W_i$
end
nc:seca2 `// Positional encoding`
X′ = X + P;
nc:positional
for *n times* **do**
  nc:encoding-block1 `// multihead attention`
  for $h = 1, \ldots, k$ **do**
    nc:multihead1 `// attention`
    Compute $Q^h = X' \cdot W_q^h$;
    nc:qh Compute $K^h = X' \cdot W_k^h$;
    nc:kh Compute $V^h = X' \cdot W_v^h$;
    nc:vh Let d_k = num columns of W_k^h;
    nc:dk Let $A^h = \frac{Q \cdot K^T}{\sqrt{d_k}}$; `// init attention function`
    nc:attention1 Let $A^h = softmax\left(A^h\right)$; `// softmax over all rows`
    nc:softmax Let $A^h = A^h \cdot V^h$; `// completed attention function`
    nc:attention2
  end
  nc:multihead2 Let $A = concatenate(A^h), h = 0, \ldots, k$;
  nc:concatenation Let $A = A \cdot W_O$; `// output projection`
  nc:outproj Let $X' = LayerNorm(X' + A)$;
  nc:addnorm1 Let $F = feedforward(X')$; `// feedforward layer`
  nc:ffn Let $X' = LayerNorm(X' + F)$;
  nc:addnorm2
end
nc:encoding-block2 Let $Z = X'$; nc:output

(*Input arguments*)

The input consists of the data series to be forecast, in our case, it is an array $s \in \mathbb{R}^n$ and of seven control hyperparameters.

For the restaurant time series, we applied a straightforward preprocessing step: *min-max* normalization of the raw data. Subsequently, each sequence was constructed using a sliding window of length $n = 7$ over the series.

Lines **??-??**: (*Input Projection*)

Single scalar values for each single timestep may not be expressive enough to capture contextual information. Each data point is projected into a higher-dimensional space where nonlinear dependencies between features can be easier to identify.

In our minimalist transformer model the projections, also known as *embeddings*, were obtained by multiplying each datapoint by a real-valued vector initialized as a random gaussian vector. These values can later be learned. In this way, a matrix $X \in \mathbb{R}^{n \times m}$ is computed, which contains, for each original timestep, its m-dimensional representation.

In the restaurant series, the projections were based on vectors of 4 values, obtaining a set of matrices, each of which with $n = 7$ rows and $m = 4$ columns, leaving 4 parameters to be learned for the projection vector. Each matrix **X** represents a window over the input time series, the first of these matrices is presented in the appendix as table **??**.

Line **??**: (*Positional Encoding*)

The basic attention mechanism, which will be implemented in lines **??-??**, is insensitive to the permutation of the input values [2] and is in itself unusable for modelling data series. Therefore, the model does not inherently deal with the sequential position of the elements: the attention only depends on the set of elements, not on their order. To inject ordering information, each input embedding X^j is added to a positional vector. This is obtained by means of a matrix $P \in \mathbb{R}^{n \times m}$, which yields the transformed input:

$$X' = X + P \tag{1}$$

where n denotes the sequence length and m the embedded dimension.

The matrix P can be either *static*, remaining fixed during the training phase, or *learnable*, changing at each step to better adapt to the task at hand. It can also be either *absolute*, depending solely on the element's position, or *relative*, in which case the element itself influences the values of the matrix's rows.

In the proposed minimalist architecture, matrix $P \in \mathbb{R}^{7 \times 4}$ is an absolute, learnable positional encoding, in the restaurant use case consisting of 7 rows and 4 columns. This equates to further 28 parameters that need to be learned. The matrix P used in our running example is presented in the appendix as table **??**.

Lines **??-??**: (*Encoding Blocks*)

The main loop of the algorithm repeats n times a block of code called an *Encoding Block*. In it, matrix **X** is first passed to a module implementing Multihead Attention (Lines **??-??**). The outputs of the heads are then concatenated and projected to keep the size consistent (Lines **??-??**). Next, a residual connection with a normalization is applied (Line **??**). The output is then passed through a small feedforward network after which another residual connection and normalization are applied (Lines **??-??**). It's important to note that each iteration uses its own set of parameters for the projections, attention weights, and feedforward layers; parameters are not shared between iterations. All these steps are detailed in the following.

Lines **??-??**: (*Multihead Attention*)

This loop iterates a basic attention mechanism that dynamically weighs the contribution of each past point in the series when computing the representation of those under scrutiny. It computes a set of attention scores relating the "query" vector of the current point and "key" vectors of all sequence values. #In our application, a query comes from the embedding of the last known observation while the keys correspond to those of all past

observations.# The resulting attention weights determine which points matter most and the forecast is then given by a weighted sum of the corresponding "value" vectors.

Queries (the elements currently focused on), keys (descriptors of the corresponding values), and values (the actual information) are all represented as matrices. Typically, queries and keys share the same inner dimension d_k, while values may have a different dimension d_v. At each iteration the corresponding set of query, key, and value is referred to as a *head*. The matrices identified for the two heads used in our running example are reported in the appendix in table **??**.

Line **??**: generates the $h-th$ query matrix and projects the input embeddings into vectors that represent what we are looking for in the past. This is achieved by using a dynamic parameter matrix $W_q^h \in \mathbb{R}^{m \times d_k}$ that is learnt.

Line **??**: generates the $h-th$ key matrix and projects the input embeddings into vectors that represent the information content of each corresponding embedding. This is achieved by using a dynamic parameter matrix $W_k^h \in \mathbb{R}^{m \times d_k}$ that is learnt.

Line **??**: generates the $h-th$ value matrix and projects the input embeddings into vectors that represent the relevant information of each available embedding. This is achieved by using a dynamic parameter matrix $W_v^h \in \mathbb{R}^{m \times d_v}$ that is learnt.

Line **??**: initializes the scaling factor.

Line **??**: implements the first part of the attention function, in this case using the *Scaled Dot-Product Attention* [2]. The similarity between sequence elements is computed from Q and K through a dot product. The key matrix is transposed so that the operation compares each feature of every query with the corresponding feature of every key. This initial A matrix will contain similarity values $a_{ij} \in \mathbb{R}$ between every query vector and every key vector.

Line **??**: transforms the similarity values into 'probability' scores. The softmax() operator is applied to all elements a_{ij} over every row: $a_{ij} = e^{a_{ij}} / \sum_k e^{a_{kj}}, \quad \forall i, j = 1, ..., n$. In this way, we get for every query vector, normalized weights of similarity towards key vectors.

Line **??**: completes the computation of the attention function by implementing a weighted sum of the value vectors, $X' = A \cdot V$, thereby aggregating information from the relevant timesteps.

Line **??**: lines **??**-**??** are repeated k times, each time obtaining an attention matrix A^h, $h = 1, \ldots, k$. This line concatenates all these matrices in a single bigger one, $A = [A^1\ A^2\ ...\ A^h]$. The matrix A obtained for the running example is reported in the appendix as table **??**.

Line **??**: since concatenation may modify the dimensionality, a final output projection $W^O \in \mathbb{R}^{hd_v \times m}$ is applied to ensure that the original size is kept $X' = A \cdot W^O$.

In our architecture, we set $h = d_k = d_v = \frac{m}{2} = 2$. This produced six matrices, all of size 4×2, totaling 48 more parameters to learn. The output matrix W^O is, instead, of size 4×4. Matrix W^O for the running example is reported in table **??** of the appendix.

Lines **??**, **??**: (*Add & Norm*)

This block tries to stabilize training by normalizing activations while keeping a residual path. It is implemented by a layer function $LayerNorm : \mathbb{R}^{n\times m} \to \mathbb{R}^{n\times m}$, that operates independently for each row i (i.e., for each embedding), computing the mean $\mu_i \in \mathbb{R}$ and the variance $\sigma_i \in \mathbb{R}$, and updating the features of row i as $x_{ij} = \gamma_j \cdot \dfrac{x_{ij} - \mu_i}{\sigma_i} + \beta_j$, where $\gamma, \beta \in \mathbb{R}^m$ are learnable parameters.

In our minimalist architecture, having set $m = 4$, and having two different *LayerNorm* blocks, 16 parameters were set to be learned. For the first layer:

$$\gamma = \begin{bmatrix} 0.8659 & 1.1768 & 0.4843 & 1.2575 \end{bmatrix} \quad \beta = \begin{bmatrix} -0.3721 & 0.1294 & -0.0496 & 0.3132 \end{bmatrix}$$

While for the second layer:

$$\gamma = \begin{bmatrix} 0.8747 & 0.7511 & 0.6320 & 0.7074 \end{bmatrix} \quad \beta = \begin{bmatrix} -0.1117 & -0.1993 & 0.0806 & 0.1641 \end{bmatrix}$$

Line **??**: (*Feedforward Network*)

The final processing step of our minimalist transformer architecture implements a two-layer network, with a ReLU activation function in between:

$$s' \leftarrow \max(0,\, s' \cdot W_1 + b_1) \cdot W_2 + b_2 \qquad \text{(Line ??)}$$

where:

$$W_1 \in \mathbb{R}^{m\times p}, \quad b_1 \in \mathbb{R}^p$$
$$W_2 \in \mathbb{R}^{p\times m}, \quad b_2 \in \mathbb{R}^m$$

The ReLU function is applied element-wise. The intermediate dimension p is typically chosen larger than m to improve model capacity and generalization.

In our model, we set $p = 4 \times m = 16$, meaning a total number of $16 \times 4 \times 2 + (16 + 4) = 148$ parameters. The values learned by the sample architecture is reported in the appendix in tables **??** and **??**.

Line **??**: (*Output*)

The full output, stored in the matrix X', contains the *encoded* representation of the input timesteps. In order to distinguish it thereafter it will be denoted as Z.

3.2. Decoding

The objective of the decoding component (i.e., the decoder) in a transformer architecture is to generate an output sequence step by step. This is achieved by using the encoded representation of the input and the output data generated by the encoding component.

The general structure of the decoding module is represented by the pseudocode in Algorithm **??**. The individual steps are detailed below.

Lines **??**-**??**: (*Input*)

The Decoder takes the encoder output $Z \in \mathbb{R}^{n\times m}$ as its main input (Line **??**) and iteratively updates a sequence represented by matrix $\mathbf{Y} \in \mathbb{R}^{o\times m}$, where each row y_i represents the i-th output embedding. Initially, the matrix is zero-initialized (Line **??**). The sequence is then updated autoregressively, i.e., one timestep at a time with each new element depending on the previously generated ones.

Lines **??**-**??**: (*Decoding Blocks*)

Algorithm 2: The decoding module

procedure decoding($\mathbf{Y}$,m,h^1,h^2)

Input : Matrix $\mathbf{Z}$ from encoding, *no* num data series points in output, l number of decoding blocks, h^1 and h^2 head numbers

Output: A 2D matrix $\mathbf{Y}$

`// subscripts indicate rows, superscripts columns`

ec:start $\mathbf{Y} = \mathbf{0}_{k \times m}$

ec:output **for** $i = 0, \ldots,$ *no* **do**

  ec:outer **for** l *times* **do**

    ec:decblock1 `// Masked self-attention`

    for $j = 0, \ldots, h^1$ **do**

      ec:self1 Compute $Q^j = \mathbf{Y} \cdot W^j_{q1}$, $K^j = \mathbf{Y} \cdot W^j_{k1}$, $V^j = \mathbf{Y} \cdot W^j_{v1}$

      ec:selfproj Let $S^j = \dfrac{Q^j {K^j}^T}{\sqrt{d_k}}$

      ec:scaled1 Set $S^j_i = -\infty$ for all $i > j$; `// mask future positions`

      ec:masking Let $A^j = softmax(S^j)V^j$ ec:softmax

    end

    ec:self2 Let $\mathbf{A} = concatenate(A^j)$, $j = 0, \ldots, h^1$

    ec:concatenation1 Let $\mathbf{A} = \mathbf{A} \cdot \mathbf{W_{o1}}$; `// for dimensionality coherence`

    ec:outproject1 Let $\mathbf{Y} = LayerNorm(\mathbf{Y} + \mathbf{A})$

    ec:norm1 `// Cross-attention`

    for $j = 0, \ldots, h^2$ **do**

      ec:cross1 Compute $Q^j = Y \cdot W^j_{q2}$, $K^j = Z \cdot W^j_{k2}$, $V^j = Z \cdot W^j_{v2}$

      ec:crossproj Let $A^j = softmax\left(\dfrac{Q^j {K^j}^T}{\sqrt{d_k}}\right)V^j$

      ec:scaled2

    end

    ec:cross2 Let $\mathbf{A} = concatenate(A^j)$, $j = 0, \ldots, h^2$

    ec:concatenation2 Let $\mathbf{A} = \mathbf{A} \cdot \mathbf{W_{o2}}$

    ec:outproject2 Let $\mathbf{Y} = LayerNorm(\mathbf{Y} + \mathbf{A})$

    ec:norm2 Let $F = feedforward(\mathbf{Y})$

    ec:ffn Let $\mathbf{Y} = LayerNorm(\mathbf{Y} + F)$

    ec:norm3

  end

  ec:decblock2 $Y_i = feedforward(Y_i) * scale(\mathbf{Z}) + bias(\mathbf{Z})$

  ec:finalproj $y_i = \text{SECA}^{-1}(Y^i)$ `// Se l'output fosse un array di scalari allora si userebbe` $SECA^{-1}$ `(che essendo ora solo un vettore indicherei come v ad esempio) ma se rimane una matrice n x m allora non si usa... Dipende da cosa intendiamo per output del decoding`

end

ec:outer2

lgo:decoding

Just as in the encoding phase, the main loop of the algorithm updates for l iterations (Lines **??**–**??**) the matrix $\mathbf{Y}$. The block of code implementing each iteration is called a *Decoding Block*. In the block, matrix $\mathbf{Y}$ is first processed by the *Multihead Masked Self-Attention* which enables the decoder to process to previous positions in the output sequence while preventing it from attending to future positions, thereby enforcing autoregressive generation. Next, $\mathbf{Y}$ interacts with the encoder output $\mathbf{Z}$ through a *Multihead Cross-Attention module* which enables the model to selectively focus on relevant parts of the input sequence (as encoded by the encoder) when generating each output token. This is the key mechanism that connects the encoder and decoder. Finally, the output is passed through some postprocessing steps to produce the final result of the single iteration.

The general procedure outlined above is explained in more detail below.

Lines **??**-**??**: (*Masked Self-Attention*)

The Masked Self-Attention mechanism works in the same way as a Multihead Attention function, but ensures that the generation of each successive sequence value can only consider the present or the past, never the future. This is achieved using a look-ahead mask that prevents feedback in time. The following operations are performed inside each loop (i.e. each head).

Line **??**: generates the query, key and value matrices, projecting the already-generated embeddings into semantic vectors. This projection is achieved by using dynamic parameter matrices, which must be learnt.

Line **??**: computes the first part of the *Scaled-Dot Product Attention* function.

Line **??**: masks out any similarity values computed between a query vector from a timestep and a key vector coming from a future timestep. During the subsequent softmax step, the value $-\infty$ becomes 0, thus avoiding dependency on the future.

Line **??**: computes the last part of the attention function.

After the loop of Lines **??**-**??** is completed, the h^1 different output matrices must be aggregated together.

Line **??**: the h^1 matrices are concatenated by row, obtaining the complete $\mathbf{A}$ matrix.

Line **??**: the matrix $\mathbf{A}$ is multiplied by $W_{o1} \in \mathbb{R}^{h^1 d_v \times m}$. The latter matrix is a learnable parameter.

The operations above are referred to as 'Self-Attention' because the projections of Q, K and V all come from the same input vector, $\mathbf{Y}$. In our example architecture, we set the number of heads for the self-attention layers to $h^1 = 2$.

Lines **??**, **??**, **??**: (*Add & Norm*) Each operation in the decoding stage is followed by a residual connection and a normalization step. The LayerNorm() function works row-wise on the embeddings, standardizing them with learnable scaling and shifting parameters (γ, β).

Lines **??**-**??**: (*Cross-Attention*)

The second Attention step in the decoding phase links the partially generated output with the encoder's representation of the input. Specifically, the decoder's current output is

projected into queries, while the encoder's output provides the keys and values. Similar to Multihead Attention, this process involves a loop (Lines **??**-**??**) and an aggregation step at the end (Lines **??**-**??**).

Line **??**: generates the query, key, and value matrices, projecting the decoder's current output into queries and the encoder's output into keys and values. Each projection uses a learnable parameter matrix to transform the embeddings into semantic vectors.

Line **??**: computes the attention function, using the already computed matrices.

As stated, the above operations are repeated h^2 times, getting h^2 attention outputs.

Line **??**: All the head's outputs are concatenated by the rows in a single matrix $\mathbf{A}$.

Line **??**: The attention matrix $\mathbf{A}$ is multiplied by $W_{o2} \in \mathbb{R}^{h^2 d_v \times m}$, which is a learnable parameter.

The name Cross-Attention is due to the fact that the output of the encoding phase Z and the decoding current output $\mathbf{Y}$ are combined, *crossing* the two matrices together. In our example, we set the number of heads of the Cross-Attention layer to $h^2 = 2$.

Line **??**: (*Feedforward Network*) The decoder applies a position-wise two-layer feedforward network with ReLU activation, as was done in the encoder:

$$\mathbf{Y} \leftarrow \max(0, \mathbf{Y} \cdot W_1 + b_1) \cdot W_2 + b_2 \qquad \text{(Line ??)}$$

Line **??**: (*Projecting the Output*)
At the end of the decoding loop, we obtain the matrix $\mathbf{Y} \in \mathbb{R}^{o \times m}$. Before mapping the j-th embedding back into its scalar prediction, Y_j is passed through a final feedforward network and regularized:

$$Y_j \leftarrow feedforward(Y_j) * scale(\mathbf{Z}) + bias(\mathbf{Z}) \qquad \text{(Line ??)}$$

For enhanced regularization purposes, a scale value and a bias value are determined through two learnable matrices W_{scale}, $W_{\text{bias}} \in \mathbb{R}^{m \times m}$, based upon the mean across the timesteps of the encoded input series $\mathbf{Z}$:

$$scale(\mathbf{Z}) = \sigma\left(W_{\text{scale}} \cdot \sum_i \frac{\mathbf{Z}_i}{d}\right), \quad bias(\mathbf{Z}) = W_{\text{bias}} \cdot \sum_i \frac{\mathbf{Z}_i}{d}$$

where σ represents the sigmoid function.

If we are at the i-th iteration of the outer output cycle (Lines **??**-**??**), the output matrix $\mathbf{Y}$ contains in position i the predicted embedding Y_i for timestep i.

3.3. Learning

The encoding and decoding processes described in sections 3.1 and 3.2 are actually included in an overarching loop that implements learning by adjusting all parameters (e.g., weights of projection matrices, attention heads, feed-forward layers, etc.) to minimize a loss function over the training data. This is typically achieved through stochastic gradient descent (SGD)-based optimization, usually backpropagation.

The process is the standard neural learning process, flowing through three main phases:

1. a *forward pass* where input sequences (time series points) are embedded into vectors, flow through layers of multi-head self-attention and feed-forward networks obtaining a forecasted value.
2. *loss computation*, where the loss function measures how far the predictions are from the ground truth. In the case of forecasting data series, the most commonly used loss functions are Mean Squared Error (MSE), Mean Absolute Error (MAE) and Huber loss, which combines the robustness of MAE with the smoothness of MSE.
3. a *backward pass* where gradients of the loss with respect to the parameters are computed by backpropagation and parameters are updated using an optimization algorithm (usually a variant of gradient descent).

The backward pass implements the actual learning process by reversing the computation flow that was used to make the forecast. A transformer's computation graph has many layers, including an embedding layer, multiple self-attention blocks (each with projections W_Q, W_K, W_V, W_O), feed-forward networks, layer norms, residual connections, etc. During backpropagation, the chain rule of calculus is applied to each layer, as with standard MLPs, and the gradient flows backwards through each operation. This means that the W_Q, W_K, W_V, W_O, feed-forward weights, embedding matrices and LayerNorm parameters all receive gradient-based updates depending on the value of the loss function.

Learning is a computation-intensive phase and greatly benefits of GPU-optimized codes. In our minimalist architecture we delegated the backward computations to the relevant classes of PyTorch [19].

The embedding matrices W_{in} and W_{out} are initialized in a way such that $W_{\text{in}} \cdot W_{\text{out}} = 1$ - to represent exactly the same projection. Specifically, W_{in} is initialized by drawing each of its elements from a uniform distribution, and W_{out} is set to be $\frac{W_I}{||W_I||^2}$. Then, during training, they are set to change without any constraint.

The training technique is as follows. The input sequence $\mathbf{X}$ of fixed length n is given to the model. The predicted element at the i-th timestep $\hat{y}_i$ is looked at, and the loss function between the ground truth element y_i is computed. The employed loss is the MSE. The backpropagation step, which updates the internal parameters of the mode, follows. Then, with probability p, the $\mathbf{Y}$ matrix is filled not with the predicted element $\hat{y}_i$ but with the effective one, y_i. The probability of that happening is gradually decreased during the training phase, in order to let the model be more reliant on its own predictions - as will it be during inference. In the case of the running example, the total number of parameters to learn in the transformer model resulted to be XXXX.

3.4. Forecasting

Subsections 3.1-3.3 describe a complete process to obtain one forecasted value. If more than one value is to be forecast, each additional forecast is appended to the end of the series, after which the process is repeated using the same fixed parameter matrices.

Figure **??** shows the results for the tiny example data series. The blue line depicts the training set, the green line the validation set, and the dashed line the model's results. Training was implemented using windows consisting of seven inputs and one output; therefore, the first seven values are not modelled. The forecast was made using seven validation values in the recursive forecasting fashion outlined above.

Despite the limited data and restrictive configuration, the transformer is highly effective in modelling the input series, though its forecasting abilities are, as would be expected, less accurate. More significant tests are reported in Section 4.

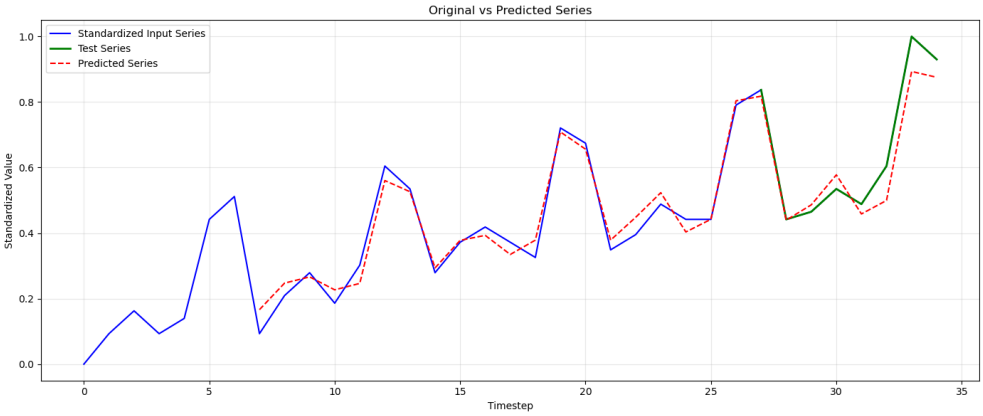

Figure 1. The Google trends restaurants dataseries
ig:restaurants

4. Computational results

This section presents results obtained using our minimalist transformer architecture in forecasting well-known univariate benchmark data series.

First successive processing stages are showcased on the archetypal airline passenger data series [5], then we present illustrative quantitative results on standard forecasting benchmark instances.

4.1. The airline passengers test case

The Airline Passengers time series is one of the most famous and widely used in data analysis and forecasting. It consists of 144 observations of the monthly total number of international airline passengers in thousands between 1949 and 1960. As it is a real-world case that is very easy to model and predict (it is known to be an ARMA(1,1)), it is commonly used as a first validation for new forecasting approaches.

When we apply our model to this series, setting aside the last 12 values for validation and accounting for the obvious 12-month seasonality, the successive encoding steps through the positional embedding - multi-head attention - layer norm - layer norm pipeline produce the results shown in Figure **??**.

The series was preprocessed only by a standard min-max normalization, then it was projected into embeddings of size $m = 12$. Employing only $n = 12$ timesteps at a time, we obtained 118 matrices $X \in \mathbb{R}^{12\times12}$. Matrix P used for positional embedding has the same dimensionality.

Next comes multi-head attention. We used 2 heads, and for each head we set $d_k = d_v = \frac{12}{2} = 6$. Therefore all matrices W_Q^h, W_K^h and W_V^h had dimensions 12×6. These provide the bases for computing the attention function, where each $h - th$ head gives rise to the matrix $A^h = \frac{Q_h K_h'}{d_h} \in \mathbb{R}^{n\times n}$, in our case $A^h \in \mathbb{R}^{12\times12}$. The successive *softmax* function does not change its dimensionality. After multiplying by V_h we obtain the output of each head, of dimension 12×6. All outputs are then concatenated by row, giving rise to a matrix $A \in \mathbb{R}^{12\times12}$. The input dimension is then kept, when the output attention matrix is multiplied by matrix $W_O \in \mathbb{R}^{hd_v\times m}$ - in our case, of dimension 12×12. In summary, the values used for the parameters were: n=12, m=12, k=2, d_k=d_v=6,p=36. The number of epochs set for feedforward network learning was 400, , the total number of parameters to learn in the transformer model resulted to be XXXX.

4.2. Benchmark results

Our work aimed to provide a basic, simple architecture that could be used to fully understand the internal functioning and possibly form the basis for improvements, rather

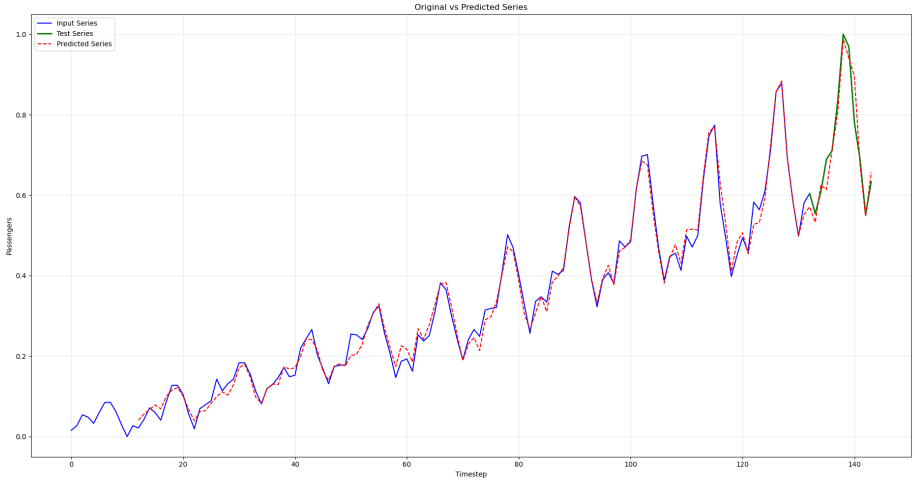

Figure 2. The predictions of the model on the full airline series, comprising 132 training data-points and 12 evaluation points.

ig:airlines

than a best-of-class architecture. However, it is important to validate its effectiveness in order to determine how representative it is of the possibilities offered by the transformer approach to data series forecasting. To this end, we ran both our system and a standard random forest forecaster, the RandomForestRegressor from scikit-learn [20], on a standard forecasting benchmark.

The benchmark used consisted of the monthly series of the M3 forecasting competition [21]. The M3 dataset includes a diverse collection of time series across different domains and frequencies, including yearly, quarterly, monthly, and other intervals. The monthly data subset includes 1428 real world time series collected at monthly intervals covering a wide range of domains, classified as:

- Macroeconomics (e.g., unemployment rates, inflation, industrial production),
- Microeconomics (e.g., sales, orders, inventory levels for individual companies),
- Industry (e.g., combined production, sales, or demand in a broader industrial sector, not just a single firm)
- Finance (e.g., stock prices, exchange rates, interest rates),
- Demographics (e.g., population statistics, birth/death rates),
- Others, i.e., diverse, real-world time series that don't belong to traditional economic or financial domains (e.g., energy consumption, transportation data)

The length of the monthly series varies, typically ranging from 48 to 126 observations (i.e., 4 to over 10 years of data) and for each series, participants were asked to forecast the next 18 months. We used this same forecast horizon in our tests, and the following argument values: In summary, the values used for the parameters were: n=24, m=36, k=4, d_k=d_v=12,p=144. The number of epochs set for feedforward network learning was 400, the total number of parameters to learn in the transformer model resulted to be XXXX.

We ran each model on all the benchmark series, computing the RMSE on both the training and test sets. To determine whether the difference in forecast effectiveness was significant, we conducted a Mann–Whitney U-test on the RMSE results for each aggregate separately. Table **??** shows the results of the comparison aggregated over the type feature. The columns show:

- *type*: type of the aggregate,
- *num*: number of series in the aggregate,
- *len*: average length of the series of the aggregate,
- *train*: number of series where the transformer had a lower RMSE on the training set,

- *test*: number of series where the transformer had a lower RMSE on the number of series where the transformer had a lower RMSE on the training set,
- *perc*: percentage of aggregate series where the transformer had a lower RMSE on the training set
- *pval*: p value of the Mann-Whitney U test.

Table 1. Minimalistic transformer and random forest, aggregate results

	type	**num**	len	**train**	**test**	**perc**	**pval**
ab:aggregate	MICRO	474	92.65	6	134	28.27	0.000
	INDUSTRY	334	140.02	3	123	36.83	0.042
	MACRO	312	130.88	3	101	32.37	0.021
	FINANCE	145	124.40	1	68	46.90	0.247
	DEMOGRAPHIC	111	123.33	3	33	29.73	0.021
	OTHER	52	82.98	0	29	55.77	0.661

The results in Table **??** show that, despite being outperformed as a modeller of the known part of the series, the transformer solution can already be competitive in forecasting, despite the limiting operational restrictions. The quality of the forecast depends on the structure of the data, resulting from dependency on the application domain. In the MICRO and DEMOGRAPHIC domains, transformers performed better than Random Forest in less than one-third of cases, whereas in FINANCE it was almost one-half, and in the OTHER domain even more than half the time. There is no statistical difference in performance between the two approaches for these last two domains, while Random Forest has a significantly lower average RMSE for the other four.

Table 2. Minimalistic transformer and random forest, aggregate results, scaled data

ab:bestworse

id	type	trtrain	trtest	rftrain	rftest	Δtrain	Δtest
N1652	MICRO	0.061	0.262	0.062	0.335	-0.001	-0.072
N1546	MICRO	0.168	0.526	0.068	0.306	0.1	0.219
N1894	INDUSTRY	0.133	0.290	0.058	0.433	0.074	-0.142
N2047	INDUSTRY	0.144	0.667	0.056	0.428	0.088	0.238
N2255	MACRO	0.068	0.088	0.021	0.494	0.046	-0.406
N2492	MACRO	0.130	0.626	0.054	0.249	0.076	0.377
N2594	FINANCE	0.074	0.597	0.015	0.737	0.059	-0.140
N2658	FINANCE	0.126	0.656	0.046	0.435	0.080	0.220
N2737	DEMOGRAPHIC	0.074	0.129	0.046	0.209	0.028	-0.079
N2758	DEMOGRAPHIC	0.214	1.007	0.034	0.310	0.180	0.696
N2817	OTHER	0.079	0.125	0.036	0.441	0.043	-0.315
N2823	OTHER	0.116	0.166	0.066	0.081	0.050	0.085

Table reftab:bestworse shows details of the instances in each category for which transformers achieved the best and worst test results compared to random forest. The columns of the table contain data that refer to scaled series and are:

- *id*: the id of the data series in the M3 collection;
- *type*: the category of the data series;
- *trtrain*: the absolute RMSE of the transformers model on the training part;
- *trtest*: the absolute RMSE of the transformers model on the test part, i.e., on the last 18 data points;
- *rftrain*: the absolute RMSE of the random forest model on the training part;

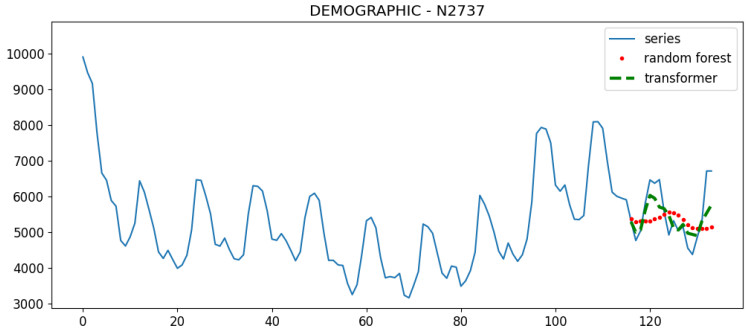

Figure 3. Actual data, transformers and random forest forecast on series N2737.
ig:trbest

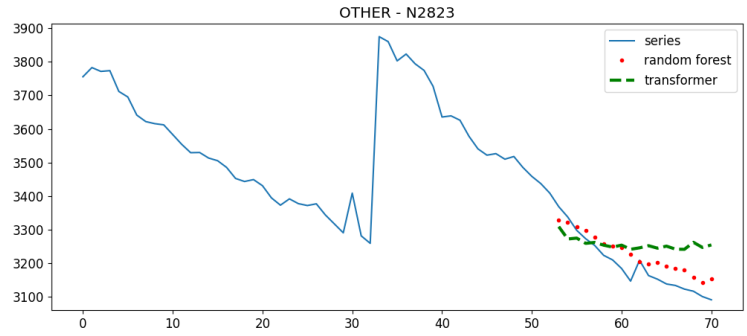

Figure 4. Actual data, transformers and random forest forecast on series N2833.
ig:trworse

- *rftest*: the absolute RMSE of the random forest model on the test part, i.e., on the last 18 data points;
- $\Delta train$: the RMSE difference between transformers and random forest on the training part;
- $\Delta test$: the RMSE difference between transformers and random forest on the test part;

The results in the table confirm that the RMSE values are comparable in all cases and that there are never differences in the order of magnitude, with values remaining relatively low when even considering that they were computed on max-min scaled series. In all categories, the transformers achieved a lower RMSE in at least one series. Figures reffig:trbest and reffig:trworse show plots of the actual series and the forecasts of the transformers and the random forest in cases where the transformers performed better and worse than the random forest, respectively.

5. Conclusions

In this work, we present a minimalist transformer architecture that is specifically designed for univariate time series forecasting. Rather than proposing yet another cutting-edge variant, our contribution lies in systematically reducing the architecture to its essential components and documenting all relevant steps of the encoder, decoder and learning procedures in pseudocode. Our aim was to provide a transparent and replicable baseline for researchers and practitioners interested in understanding the core mechanisms of transformers when applied to forecasting tasks.

We evaluated our model using a set of standard forecasting benchmarks and compared its performance with that of a random forest forecaster. Although the minimalist transformer does not outperform the most sophisticated state-of-the-art models, it consistently

achieves comparable accuracy, particularly for time series with long-term dependencies, where attention mechanisms can exploit global context more effectively than purely tree-based methods.

Beyond raw predictive performance, our findings highlight two key outcomes.

Clarity and accessibility. By explicitly exposing each functional step, our framework lowers the barrier for newcomers to the transformer literature and the field of time series forecasting.

The robustness of the transformer paradigm is also evident, as even a stripped-down version of the model demonstrates stable forecasting ability. This suggests that many performance gains in advanced variants stem from relatively minor architectural refinements.

In a research landscape often dominated by increasingly complex architectures, our study highlights the importance of transparent baselines. Future work may explore the impact of specific enhancements, such as seasonal embeddings, multi-head configurations or hybridization with statistical components, on forecasting performance when added to this minimalist core.

Appendix A

This appendix contains all values of the trained matrices obtained for the encoding of the running example described in section 3.

Table **??** shows the first embedded matrix, X, which encodes the input series. The matrix has been transposed for the sake of compact presentation.

Table A1. The first 7×4 embedded matrix X used in the encoding process.

ab:matX
$$X^T = \begin{bmatrix} 0.7388 & 0.8208 & 0.8823 & 0.8208 & 0.8618 & 1.1283 & 1.1898 \\ 0.1354 & 0.0672 & 0.0160 & 0.0672 & 0.0331 & -0.1887 & -0.2399 \\ 0.4822 & 0.5630 & 0.6237 & 0.5630 & 0.6035 & 0.8663 & 0.9269 \\ -0.1412 & -0.1238 & -0.1107 & -0.1238 & -0.1151 & -0.0585 & -0.0454 \end{bmatrix}$$

Table **??** shows the positional encoding matrix, P. The matrix has been transposed for the sake of compact presentation.

Table A2. The 7×4 positional encoding matrix P.

ab:matP
$$P^T = \begin{bmatrix} -0.6293 & -0.7571 & -0.4988 & -0.9708 & -1.1165 & 0.4899 & 0.5061 \\ 0.3700 & -0.1533 & -0.2298 & 1.0685 & 0.2047 & 1.1688 & 0.1518 \\ 0.4210 & 0.4065 & -0.2935 & -0.2282 & -1.0394 & -0.9300 & -1.2985 \\ 0.8584 & -1.0690 & 0.3565 & -0.6471 & 0.9570 & 0.8161 & -1.0734 \end{bmatrix}$$

Table **??** presents the query matrices W_q^1 and W_q^2, the key matrices W_k^1 and W_k^2 and the value matrices W_v^1 and W_v^2 for the two heads used in the restaurant running example.

Table A3. Query, key and value matrices.

ab:matWq
$$W_q^1 = \begin{bmatrix} 0.596 & -1.110 & 1.512 & 0.011 \\ 0.768 & -0.201 & -0.149 & 0.941 \end{bmatrix} \quad W_q^2 = \begin{bmatrix} 0.182 & 1.118 & 0.137 & -0.102 \\ 0.166 & -0.413 & -1.276 & 1.259 \end{bmatrix}$$
$$W_k^1 = \begin{bmatrix} 1.339 & -0.812 & 1.077 & 0.332 \\ -1.298 & -0.333 & -0.304 & 0.378 \end{bmatrix} \quad W_k^2 = \begin{bmatrix} -1.427 & 0.111 & -0.207 & -0.5870 \\ 1.485 & 0.555 & 0.561 & -1.182 \end{bmatrix}$$
$$W_v^1 = \begin{bmatrix} 1.339 & -0.812 & 1.077 & 0.332 \\ -1.298 & -0.333 & -0.304 & 0.378 \end{bmatrix} \quad W_v^2 = \begin{bmatrix} 0.016 & -0.613 & 0.161 & -0.264 \\ -0.440 & 1.467 & -0.990 & -0.360 \end{bmatrix}$$

Table **??** present the attention matrix as resulting after the multihead attention phase for the restaurant time series.

Table **??** shows the matrix used to recover the needed dimensionality after the multihead attention computation.

Table A4. The matrix A obtained.

ab:matA
$$A = \begin{bmatrix} -0.0722 & -0.0876 & -0.0739 & -0.0933 & -0.0885 & -0.0770 & -0.1200 \\ -0.6445 & -0.6210 & -0.6460 & -0.6112 & -0.6374 & -0.6563 & -0.5435 \\ -0.6498 & -0.6190 & -0.6459 & -0.6032 & -0.6224 & -0.6393 & -0.5541 \\ 0.3640 & 0.3344 & 0.3603 & 0.3215 & 0.3361 & 0.3554 & 0.2753 \end{bmatrix}$$

Table A5. The matrix W^O used for dimensionality reduction of the running example.

ab:matWO
$$W^O = \begin{bmatrix} 0.5883 & 0.2314 & -0.3120 & 0.9704 \\ -0.2597 & -0.2973 & 0.0408 & -0.3421 \\ -0.0065 & -0.4276 & -0.2859 & -1.0913 \\ -0.0770 & 0.7207 & 0.0433 & -0.1555 \end{bmatrix}$$

Tables **??** and **??** show the weights learnt by the feed-forward neural network used at the end of the encoding phase. They show the weights of the hidden and output layers, respectively.

Table A6. Feedforward network, weights of the internal layer.

ab:matW1b1
$$W_1^T = \begin{bmatrix} 0.4196 & -0.8769 & 0.3610 & -0.0478 \\ -0.0640 & -0.0074 & 0.1623 & 0.0387 \\ -0.5033 & 0.0526 & 0.4721 & -0.0462 \\ -0.0235 & -0.0180 & -0.0918 & 0.0166 \\ -0.1677 & 0.0475 & 0.1042 & -0.0290 \\ -0.3696 & -0.4210 & 0.5562 & 0.1345 \\ 0.1058 & -0.1836 & -0.0906 & -0.1468 \\ 0.2463 & 0.2171 & -0.0282 & 0.0412 \\ 0.2532 & 0.2173 & -0.0565 & 0.2351 \\ -0.0807 & -0.2376 & 0.1662 & 0.0452 \\ -0.4880 & -0.2997 & 0.1987 & 0.0231 \\ 0.1958 & -0.0170 & -0.2829 & -0.0767 \\ 0.0399 & -0.0465 & -0.0626 & 0.0403 \\ 0.2329 & 0.2049 & -0.1548 & -0.0405 \\ -0.3543 & -0.4732 & 0.1578 & 0.3250 \\ -0.1712 & -0.0503 & 0.0477 & -0.1804 \end{bmatrix} \quad b_1 = \begin{bmatrix} -0.6151 \\ -0.3014 \\ -0.5325 \\ -0.2021 \\ -0.4317 \\ -0.1031 \\ -0.5842 \\ -0.1081 \\ -0.2168 \\ -0.6052 \\ -0.0827 \\ -0.4561 \\ -0.5228 \\ -0.3175 \\ -0.2039 \\ -0.4153 \end{bmatrix}$$

Table A7. Feedforward network, weights of the output layer.

ab:matW2b2

$$W_2 = \begin{bmatrix} 0.4476 & -0.4134 & -0.0992 & 0.3374 \\ -0.2983 & -0.0121 & -0.0738 & -0.0373 \\ -0.2697 & -0.2605 & 0.2736 & 0.1703 \\ 0.0031 & 0.1685 & -0.0202 & 0.2742 \\ -0.2344 & 0.0643 & 0.1967 & -0.0713 \\ 0.0128 & -0.0973 & -0.2267 & 0.3215 \\ -0.0078 & -0.1763 & 0.1047 & 0.0694 \\ 0.0417 & -0.0061 & 0.1713 & -0.1046 \\ -0.0631 & 0.2215 & -0.0927 & -0.1762 \\ -0.0458 & -0.2141 & -0.0208 & 0.0600 \\ -0.1016 & -0.1437 & -0.0314 & 0.2947 \\ 0.0008 & -0.1836 & 0.0040 & 0.0332 \\ -0.0714 & 0.0968 & -0.2715 & 0.1422 \\ 0.0879 & 0.0680 & 0.0728 & -0.2513 \\ 0.3400 & -0.1222 & -0.0918 & 0.4382 \\ -0.0747 & -0.0262 & -0.0424 & 0.0557 \end{bmatrix} \quad b_2 = \begin{bmatrix} -0.0854 \\ 0.0029 \\ -0.0093 \\ 0.1207 \end{bmatrix}$$

Author Contributions: All authors contributed equally to the work and have read and agreed to the published version of the manuscript.

Funding: This research received no external funding

Data Availability Statement: In this section, please provide details regarding where data supporting reported results can be found, including links to publicly archived datasets analyzed or generated during the study.Suggested Data Availability Statements are available in section "MDPI Research Data Policies" at https://www.mdpi.com/ethics.

Acknowledgments: No generative AI has been used for purposes such as generating text, data, or graphics, or for study design, data collection, analysis, or interpretation of data. AI-assisted tools were only used to check the spelling, syntax and style of the writing.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

MDPI	Multidisciplinary Digital Publishing Institute
DOAJ	Directory of open access journals
TLA	Three letter acronym
LD	Linear dichroism

1. Cun, Y.L.; Bottou, L.; Bengio, Y. Reading checks with multilayer graph transformer networks. In Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing, 1997, Vol. 1, pp. 151–154 vol.1.
2. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.u.; Polosukhin, I. Attention is All you Need. In Proceedings of the Advances in Neural Information Processing Systems; Guyon, I.; Luxburg, U.V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; Garnett, R., Eds. Curran Associates, Inc., 2017, Vol. 30.
3. Hochreiter, S.; Schmidhuber, J. Long Short-Term Memory. *Neural Computation* **1997**, *9*, 1735–1780.
4. Chung, J.; Gulcehre, C.; Cho, K.; Bengio, Y. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, 2014. cite arxiv:1412.3555Comment: Presented in NIPS 2014 Deep Learning and Representation Learning Workshop.
5. Box, G.; Jenkins, G. *Time Series Analysis: Forecasting and Control*; Holden-Day series in time series analysis and digital processing, Holden-Day, 1970.
6. Holt, C.C. Forecasting trends and seasonals by exponentially weighted moving averages. *ONR Memorandum* **1957**, *52*, 5–10.
7. Winters, P.R. Forecasting Sales by Exponentially Weighted Moving Averages. *Management Science* **1960**, *6*, 324–342.
8. Zhou, H.; Zhang, S.; Peng, J.; Zhang, S.; Li, J.; Xiong, H.; Zhang, W. Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting, 2021, [arXiv:cs.LG/2012.07436].
9. Wu, H.; Xu, J.; Wang, J.; Long, M. Autoformer: Decomposition Transformers with Auto-Correlation for Long-Term Series Forecasting, 2022, [arXiv:cs.LG/2106.13008].
10. Zhou, T.; Ma, Z.; Wen, Q.; Wang, X.; Sun, L.; Jin, R. FEDformer: Frequency Enhanced Decomposed Transformer for Long-term Series Forecasting. In Proceedings of the Proceedings of the 39th International Conference on Machine Learning; Chaudhuri, K.; Jegelka, S.; Song, L.; Szepesvari, C.; Niu, G.; Sabato, S., Eds. PMLR, 17–23 Jul 2022, Vol. 162, *Proceedings of Machine Learning Research*, pp. 27268–27286.
11. Nie, Y.; Nguyen, N.H.; Sinthong, P.; Kalagnanam, J. A Time Series is Worth 64 Words: Long-term Forecasting with Transformers, 2023, [arXiv:cs.LG/2211.14730].
12. Wen, Q.; Zhou, T.; Zhang, C.; Chen, W.; Ma, Z.; Yan, J.; Sun, L. Transformers in time series: a survey. In Proceedings of the Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, 2023, IJCAI '23.

13. Ahmed, S.; Nielsen, I.; Tripathi, A.; Siddiqui, S.; P., R.R.; G., R. Transformers in Time-Series Analysis: A Tutorial. *Circuits Syst Signal Processing* **2023**, *42*, 7433–7466.
14. Su, L.; Zuo, X.; Li, R.; Wang, X.; Zhao, H.; Huang, B. A systematic review for transformer-based long-term series forecasting. *Artificial Intelligence Review* **2025**, *58*, 1573–7462.
15. Lim, B.; Sercan, O.; Loeff, N.; Pfister, T. Temporal Fusion Transformers for interpretable multi-horizon time series forecasting. *International Journal of Forecasting* **2021**, *37*, 1748–1764.
16. Liu, Y.; Hu, T.; Zhang, H.; Wu, H.; Wang, S.; Ma, L.; Long, M. iTransformer: Inverted Transformers Are Effective for Time Series Forecasting. In Proceedings of the The Twelfth International Conference on Learning Representations, 2024.
17. Garagnani, F. deconstructing-transformers, 2025. https://github.com/FGaragnani/deconstructing-transformers, last accessed in 08.08.2025.
18. Google. Google Trends. https://trends.google.com, 2025. Accessed: 2025-08-27.
19. PyTorch Transformer, 2025. https://docs.pytorch.org/docs/stable/generated/torch.nn.Transformer.html, last accessed in 08.08.2025.
20. Geurts, P.; Ernst, D.; Wehenkel, L.
21. Makridakis, S.; Hibon, M. The M3-Competition: results, conclusions and implications. *International Journal of Forecasting* **2000**, *16*, 451–476. The M3- Competition.