ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Artificial Intelligence in Industry

TRANSFORMERS ARCHITECTURES FOR TIME SERIES FORECASTING

CANDIDATE
Andrea Policarpi

SUPERVISOR
Prof. Michele Lombardi

CO-SUPERVISORS
Dr. Rosalia Tatano
Dr. Antonio Mastropietro

Academic year 2020-2021

Session 3rd

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Time series forecasting is an important task related to countless applications, spacing from anomaly detection to healthcare problems. The ability to predict future values of a given time series is a non-trivial operation, whose complexity heavily depends on the number and the quality of data available. Historically, the problem has been addressed first by simple, statistical models, and later by deep learning-based models such as *convolutional* and *recurrent neural networks*. Since the 2018's publication of the *Transformer*, various transformer-based models managed to achieve state-of-the-art results in various fields, including the forecasting of time series; in this context, many model proposals can be found in the literature, each with its own uniqueness. Starting from this, the work presented in this thesis aims to achieve two main objectives:

- Apply two transformer-based models, namely a *TransformerT2V* and an *Informer*, to two different time series forecasting problems, and compare the results with the ones obtained by two non-transformer architectures, represented by a *CNN* and a *LSTM*;
- Investigate the internal mechanisms behind the *Informer*'s key component, the *Probsparse attention*, and suggest some improvements in order to further enhance the model's performances.

Regarding the first point, the models have been trained on two public datasets, namely the *ETTm1* and *CU-BEMS*, and their performances have been evaluated both in qualitative and quantitative terms; for the second goal, the focus of the experiments has instead been on the hyperparameter responsible for the degrees of the approximations carried out by the Probsparse mechanism, which have been quantified and evaluated by means of appropriate metrics.

This thesis is structured as follows: Chapters 2 and 3 introduce the topic background and related work present in the literature, while Chapters 4 and 5 describe in the detail the datasets and the architectures involved in the investigations. Chapter 6 illustrates the performed experiments and the methodology followed for their execution, while their results are provided in Chapter 7. Finally, Chapter 8 is reserved for some final remarks and possible future work suggestions.

Chapter 2

Background

2.1 Forecasting and Time Series

The act of forecasting is vital for many scientific and non-scientific activities. A scientist would like to predict the behaviour of a given system, in order to understand its mechanisms and exploit its properties; likewise, a financial economist is surely interested in anticipating the market trends, in order to make a profit from it. A successful epidemiologist is able to predict the spread of an infectious disease as a function of different courses of action, and so on.

The most common form of forecasting involves *time series*. A time series, in short, is an ordered sequence of values of one or more variables at successive points in time. If the values are distributed at equally spaced time intervals, the time series is *regular*, and, given the starting time and the timestep between two consecutive values, the series X can be written, without loss of information, with the notation:

$$X = x_0, x_1, x_2, ..., x_{t-1}, x_t, x_{t+1}, ... \tag{2.1}$$

where x_t is the value of the variable(s) of the series at timestep t. An example of time series is depicted in Fig. 2.1.

A time series is *univariate* if it contains a single time-dependent variable,

or *multivariate* if more than one. Almost everything that is measurable can be collected into a time series. Some examples of time series:

- The ECG signal of a patient;
- The retail sales of a product;
- The temperature and relative humidity inside a building;
- The daily electrical consumption of an office;
- The weekly number of taxi calls in a city.

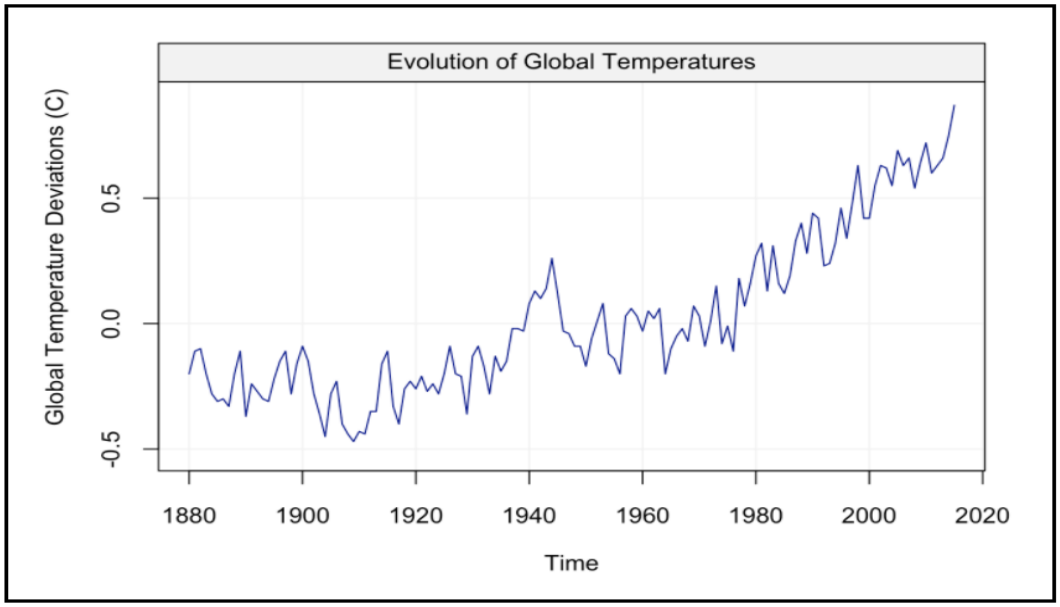


Figure 2.1: Example of time series for global temperature deviation.

The information content of a time series is usually the result of multiple underlying patterns, and it is often useful to recognize and extract these patterns in order to have a better understanding of the data. Overall, time series can be seen as a sum of three major components: *trend*, corresponding to the meaningful non-periodic information, *seasonality*, representing the periodic information, and *residual*, enclosing the noise components (Fig.2.2).

The trend reflects a long-term increase or decrease in the data, not necessairly linear [16]; it reflects the overall direction of the series, net of local oscillations. The latter are instead included in the seasonal component of the series: recurrent behaviors dictated by periodic conditions or events such as a certain time of the day or a month of the year. Seasonality is always of a fixed and known frequency [16]; if multiple patterns occur at different frequencies in the same series, the dominant one is taken into account. As for the residual component, it collect the remainder of the series that is neither trend or seasonal: mostly noise and irregular fluctuations, and sometimes minor recurring behaviors with different frequencies with respect to the seasonal one.

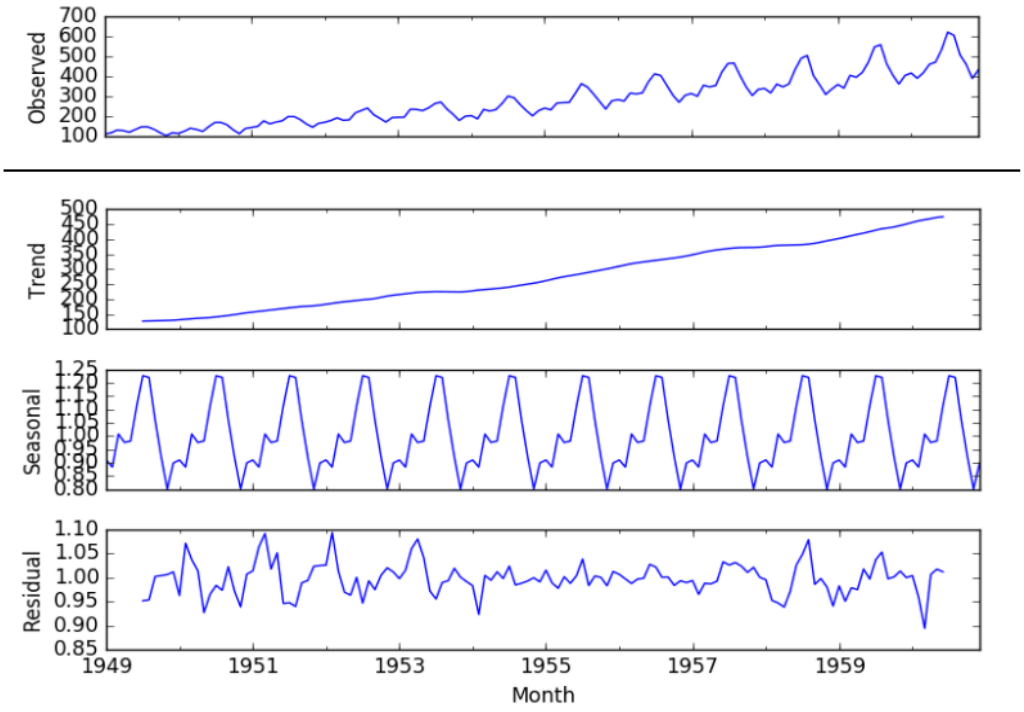


Figure 2.2: A time series and its decomposition into its three main components (Image from [4]).

The composition of these three components into the original series can either be *additive* or *multiplicative* [16]. For each element y_t of a series Y, the additive composition takes the form:

$$y_t = T_t + S_t + R_t \tag{2.2}$$

while the multiplicative compositions is of the form:

$$y_t = T_t \cdot S_t \cdot R_t \tag{2.3}$$

where T_t, S_t and R_t are the elements of trend, seasonality and residual components of the series for each time step t. The main difference between the two situations resides in the fact that in the latter the magnitude of the periodic oscillations is proportional to the series level, while in the additive case this does not hold and the magnitude is often constant.

Given the pervasive nature of time series, the act of time series forecasting holds countless applications. In order to describe them, it is first necessary to formalize the problem.

2.1.1 The Time Series Forecasting Problem

Time series forecasting, in short TSF, can be carried out in many ways, and some classification can be made [16][26]. Forecasting problems differ by:

- **The prediction object.** In the *point estimates* case we predict the expected future values of a target variable, while in the *probabilistic forecasting* case we obtain the parameter values of a distribution of probability (e.g. Gaussian) associated to it (useful to take into account the model's uncertainty);

- **The forecasting window.** Depending on the model output, the forecasting can either be *one-step ahead*, or *multi-horizon* (in the latter, a window of M timesteps in the future is predicted simultaneously);

- **The input and output dimensionality.** Input and output time series can either be *univariate* or *multivariate*, thus enabling various combinations. For example, in a *multi-to-single* forecasting, from past values

of a multivariate time series we try to infer future values for an univariate one.

Taking into account the point estimates case, the univariate one-step ahead forecasting can be formalized as follows:

Definition 1. *Let Y be an univariate time series for a target variable y. For the current time step t, the last k values of T are:*

$$y_{t-k+1}, y_{t-k+2}, ..., y_{t-1}, y_t \tag{2.4}$$

We define the one-step ahead forecasting of the series T at time t over a lookback window k as the prediction of the next y_{t+1} value of the series, as a function of its last k values:

$$\hat{y}_{t+1} = f(y_{t-k+1}, y_{t-k+2}, ..., y_{t-1}, y_t) \tag{2.5}$$

where $\hat{y}_{t+1}$ is the predicted value of y_{t+1}.

The function $f(\cdot)$ is model-dependent, and can vary from simple to very complex depending on the input elaborations taken into account.

The provided definition can be easily extended to the multi-horizon case by considering a certain *forecasting window* M. Furthermore, the multi-to-single forecasting case is covered by introducing the concept of *covariate* time series, additional series used to help explaining the target one. We have:

Definition 2. *Let T be a multivariate time series, composed by an univariate series Y for a target variable y and N univariate series $X^1, X^2, ..., X^N$ associated to some auxiliary variables $x^1, x^2, ..., x^N$. If Y is the target of forecasting, the series $X^1, X^2, ..., X^N$ are called covariate time series for Y in T. Let $x_{A:B}$ be a short notation for $x_A, x_{A+1}, ..., x_B$.*

We define the M-horizon forecasting of the target series Y at time t over a lookback window k as the prediction of the next M values of the series Y, as a function of the last k values of Y and $X^1, X^2, ..., X^N$:

$$[\hat{y}_{t+1}, ..., \hat{y}_{t+M}] = f(y_{t-k+1:t}, x^1_{t-k+1:t}, ..., x^N_{t-k+1:t}) \tag{2.6}$$

where $\hat{y}_{t+1}, ..., \hat{y}_{t+M}$ *are the predicted values of* $y_{t+1}, ..., y_{t+M}$.

The simplest way to approach a multi-horizon forecasting is by iteratively applying a "one-step ahead" model, that for each $j \in \{1, 2, ..., M\}$ takes as input the past $y_{t-k+j}, ..., y_{t+j-1}$ and $x^i_{t-k+j}, ..., x^i_{t+j-1}$ values of the series (and using past predictions for the timesteps $t+1, ..., t+j-1$) and predicts the next y_{t+j} element of the target series.

Other methods prefer instead to predict all the values in the horizon at the same time, thus relying only on past values of the series, without taking into account intermediate predictions which could be incorrect.

Definition 2 can be further extended in order to take into account the multivariate output case, and analogue definitions, although structurally different, can be made for the probabilistic forecasting problem. These formalizations will be here omitted, as this thesis work is focused onto the point estimates, multi-to-single time series forecasting problem.

2.1.2 Applications

Due to its pervasivity, the TSF problem is related to countless applications [16][26]. Well-performing TSF methods and architectures would be beneficial for:

- **Anomaly detection.** The predicted values of a time series related to a given system can be interpreted as the expected future behaviour of the system itself. By comparing the expected and the real behaviour, we can spot and quantify the occurrence of *anomalies*, and send an *alarm signal* if the anomaly falls over a given threshold.

- **Epidemic scenarios forecasting.** The forecasting of epidemic time series can be exploited not only to study the evolution of the disease, but

also to simulate scenarios: if a given starting state and a disease response strategy are correctly encoded in the input series, the predicted output can be studied to evaluate the effectiveness of the strategy with respect to the disease evolution.

- **Economic domain problems.** Many economic problems, such as stock market prediction and portfolio management, can be directly reformulated as TSF problems.

- **Resource optimization and scheduling.** By forecasting the need of given resources over time, it is possible to allocate them efficiently. This includes scheduling problems, on which the resource to optimize is represented by time.

- **System evolution forecasting.** The forecasting can be used to predict the evolution of certain systems of interests; a classical example is the atmospheric system in the weather forecasting problem.

2.1.3 Challenges of the TSF problem

Predicting the future involves dealing with the uncertain and the unknown. In the time series forecasting problem, the major complication is given by the fact that predictions far into the future often resemble the behaviour of chaotic systems: given a small perturbation of the initial state (in our case, the input series), the output forecasts may differ very significantly. Extending the forecasting window causes some degree of error accumulation; the farther we try to graze into the future, the lower our accuracy will be (Fig.2.3).

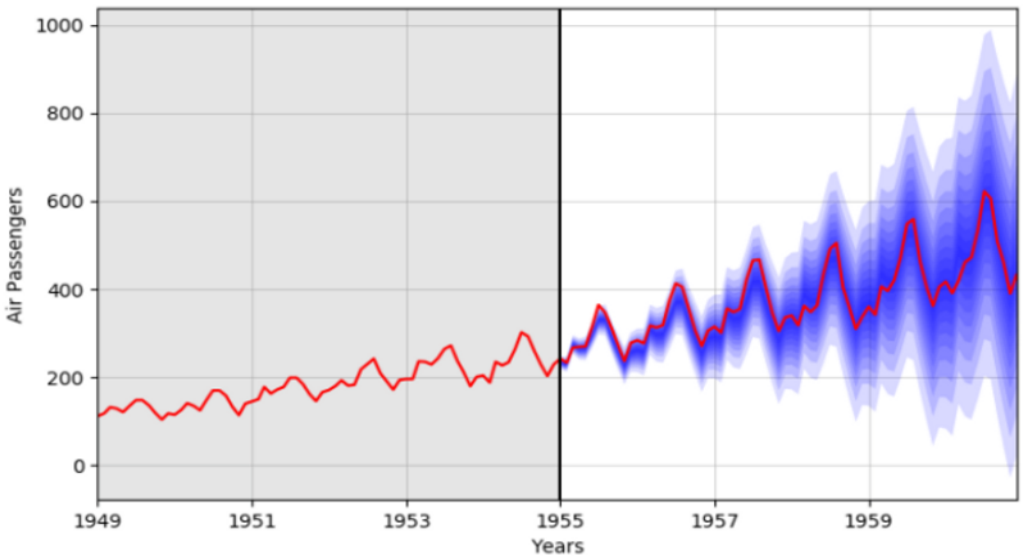


Figure 2.3: Visualization of how expanding the forecasting horizon entails a progressive decrease in accuracy (Image from [13]).

Furthermore, not every forecasting problem is equally difficult: for instance, in the same 24h timespan, the indoor temperature shift is much easier to predict than the air pollution of the city. Depending on the application, the problematic aspects of TSF can be traced back to some sort of *weakness*, either in the *data* or in the *model* used for the forecasting.

As for the **data**, the major issues are represented by *data scarcity*, *missing values* and *noise*. The first occurs when the time series dataset used to train the forecasting model is too small to achieve acceptable results; for many applications the historical data is not available or difficult to achieve, thus resulting in poor training datasets. A similar problem holds for *missing values*, namely timesteps of the series for which we don't have a value: while they can be handled by means of several methods (moving average interpolation, last-observed propagation, etc.), the guess is never exact and their presence is source of degradation in performances. As for the *noise*, typically the values of a time series present some sort of white noise (a disturbance component

which is uniformly distributed and zero-centered), plus some additional interferences, depending on the series. When these elements are not negligible, the overall forecasting process is hindered.

For what regards the **model** side, two major critical issues reside in the difficulty to extract *long-term dependencies* in the time series and to handle long input and output *timestep windows*. Many forecasting models present these weaknesses, due to the lack of mechanisms to elaborate in a meaningful way long input sequences and their internal correlaton.

2.2 History of models used for the TSF problem

In the past, various mathematical models have been adopted to tackle the time series forecasting problem. The works of Lim et al.[44] and Green et al.[26] provide a summary of the historically most important ones, together with their strengths and weaknesses. To define some taxonomy, it can be said that the most recent models fall into one of these three categories: *statistical*, *machine learning/deep learning based* and *deep learning based with attention mechanism*. For what concerns this work, we instead make a distinction between *Non-Transformer based* and *Transformer based* models. The rationale behind this choice is due to the fact that while architectures falling in the first category (which includes both statistical and machine learning/deep learning based models) have historically been relevant and are still widely used for TSF applications, recent scientific works are focused on the *Transformer* model and its variations [39][44], due to their ability to outperform previous models [27]. Many state of the art architectures used in TSF are also transformer-based [25][28][47].

2.2.1 Non-Transformer based models

The most used non-machine learning approach to TSF is the **Autoregressive Integrated Moving Average** (ARIMA) model. It is a statistical model that

employs the idea of moving averages to learn the serial correlation of the series (namely, the correlation between the series and a lagged version of itself). ARIMA models are a result of three components:

- Autoregressive (AR), which contribution to the output is a linear combination of past values of the series;
- Integrated (I), which is in charge of differencing consecutive values in order to make the series *stationary*;
- Moving Average (MA), which exploit past forecast errors in a regression-like model as a contribution to the output.

A full ARIMA(p,d,q) model can be written as [16]:

$$y'_t = c + \phi_1 y'_{t-1} + ... + \phi_p y'_{t-p} + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q} + \epsilon_t \tag{2.7}$$

where y'_t is the series differenced d times, ϕ and θ are the model parameters, ϵ is the white noise, p is the order of the autoregressive part and q the order of the moving average part.

While these models are easy to implement and computationally inexpensive, representing a good tool for low-complexity forecasting applications, they struggle to grasp input dependencies in more difficult problems: they provide a ”black-box” approach, in which the output is computed purely from the input data, without a meaningful elaboration of the underlying system’s state [26].

Moving on to deep learning-based models, a major representative is the class of **Convolutional neural networks.** This class of neural networks, originally created to analyze image inputs, can be adapted to the elaboration of

time series [21][26]. The peculiarity of CNNs resides in the use of *convolutional layers*, which are able to analyze not the single input values, but windows of them, by means of sliding filters (bidimensional for images, monodimensional for time series). With this mechanism, a CNN model is able to learn short-term dependencies between a time step and its neighbours. In TSF applications, in order to consider only past correlations (since we don't know future values in advance), the standard convolution is replaced by a *causal* convolution, in which only the past neighbours are considered for each input element (Fig.2.4, Fig.2.6a).

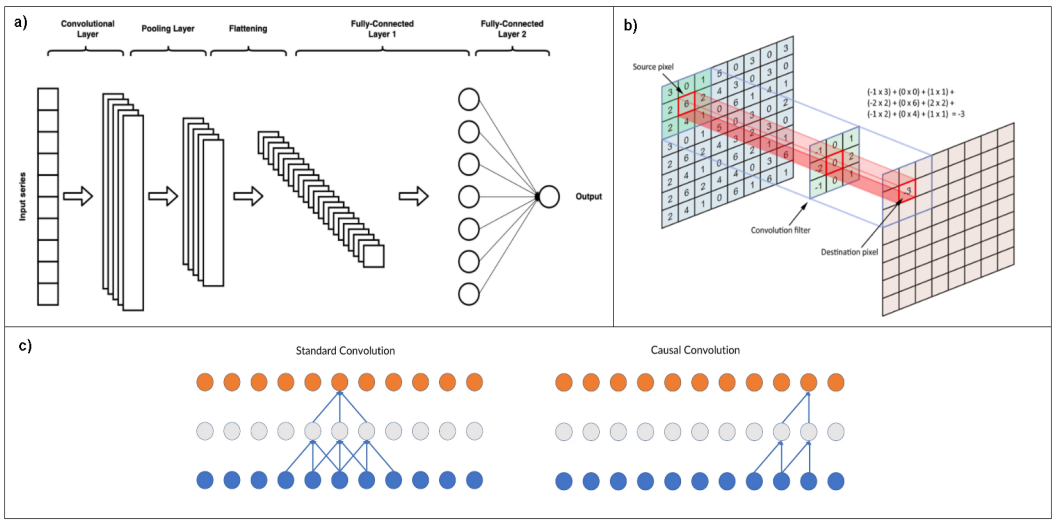


Figure 2.4: (a): Example of convolutional neural network architecture for time series forecasting (Image from [23].) (b): 2D convolution with a 3x3 filter (Image from [10]). (c): Difference between standard and causal convolution (Image from [20]).

Convolutional neural networks come with two main weaknesses. First of all, by using the same set of filter weights at each time step, they assume that input dependencies are *time-invariant*: in this hypothesis, it is taken for granted that the relation between two input elements only depends on their relative distance and not on their absolute position in the sequence. Secondly, the filter size K determines the network's ability to handle these correlations.

Distant correlations require very long filters, resulting in a cost on memory and computational efficiency. This can be partially tackled by using *dilated convolutions*, albeit at the cost of a lower output accuracy.

Another widely used group of deep learning architectures is the class of **Recurrent neural networks** (RNNs): due to the sequential structure of time series, the use of RNNs has been proven beneficial for TSF problems [7][26]. A recurrent layer is characterized by the ability to store into the *memory* of its units some of the information related to the input passing through the network. The memory state of each cell is recursively updated at each time step, thus keeping track of the previous values of the time series while analyzing the current one (Fig.2.6b). Each recurrent layer can be seen as an infinite multi-layer dense network that keeps reusing the same weights; thus we can provide an unfolded visualization for it (Fig.2.5a).

Due to this infinite lookback window, the original RNN units suffered from the so-called *vanishing/exploding gradient* problem: by propagating through multiple "equivalent" layers, during the backpropagation step of the training the gradients tend to shrink/grow exponentially, thus making the network unable to learn long-range dependencies in the data. While the use of optimized units such as LSTM (Fig.2.5b) and GRU has greatly reduced this issue, the inefficiency of RNNs to handle long inputs still represents one of their major weaknesses.

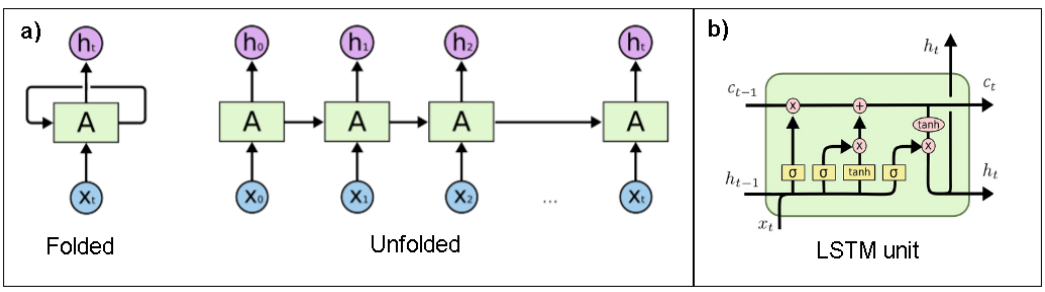


Figure 2.5: (a): Recurrent layer in its folded (left) and unfolded (right) forms. (b): Internal structure of a LSTM unit (Images from [30]).

The unexpected effectiveness of Transformer architectures in almost all machine learning problems [2][27][45] has given the push to upgrade non-transformer architectures with transformer components. One of the hybrid architectures thus generated is the **RNN enhanced with an attention layer**[26]. Adding the attention mechanism to the network has shown significant improvement in tasks involving long input sequences, such as in the TSF case. Attention layers aggregate input timestep values by means of dynamically generated weights (Fig.2.6c), allowing the network to keep track of distant time steps and their correlation with near ones.

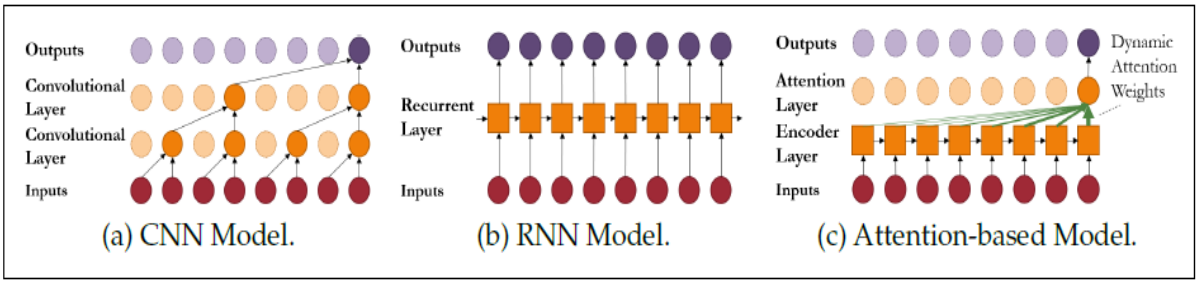


Figure 2.6: Input elaboration pipeline for the CNN, RNN and Attention-based models (Image from [26]).

2.2.2 The SOTA: Transformer-based models

Transformer based models are complex models able to achieve State-of-The-Art performances in various machine learning problems, and time series forecasting is one of them. All the architectures falling in this category are a derivation of the original Vaswani et al. *Transformer* model [42]. While some of their components may be substantially different, they all share three key elements, namely:

- A positional/temporal embedding;
- A multilayer encoder-decoder body;
- A multi-head attention mechanism.

The following paragraph will describe the original *Transformer* architecture and its main components. A summary of the current SOTA models for the TSF problem will be included in Chapter 3.

The Transformer model

The Transformer is a multi-purpose model, and can be adapted to handle inputs and outputs corresponding to different interpretations. The overall scheme is depicted in Fig.2.7a. One of its peculiarities is represented by the fact that both the *encoder* and the *decoder* accept an input: the encoder one is processed and subsequently combined with the decoder's by means of the attention mechanism. The encoder receives the proper input, while the decoder takes past output values in order to keep a trace of past elements.

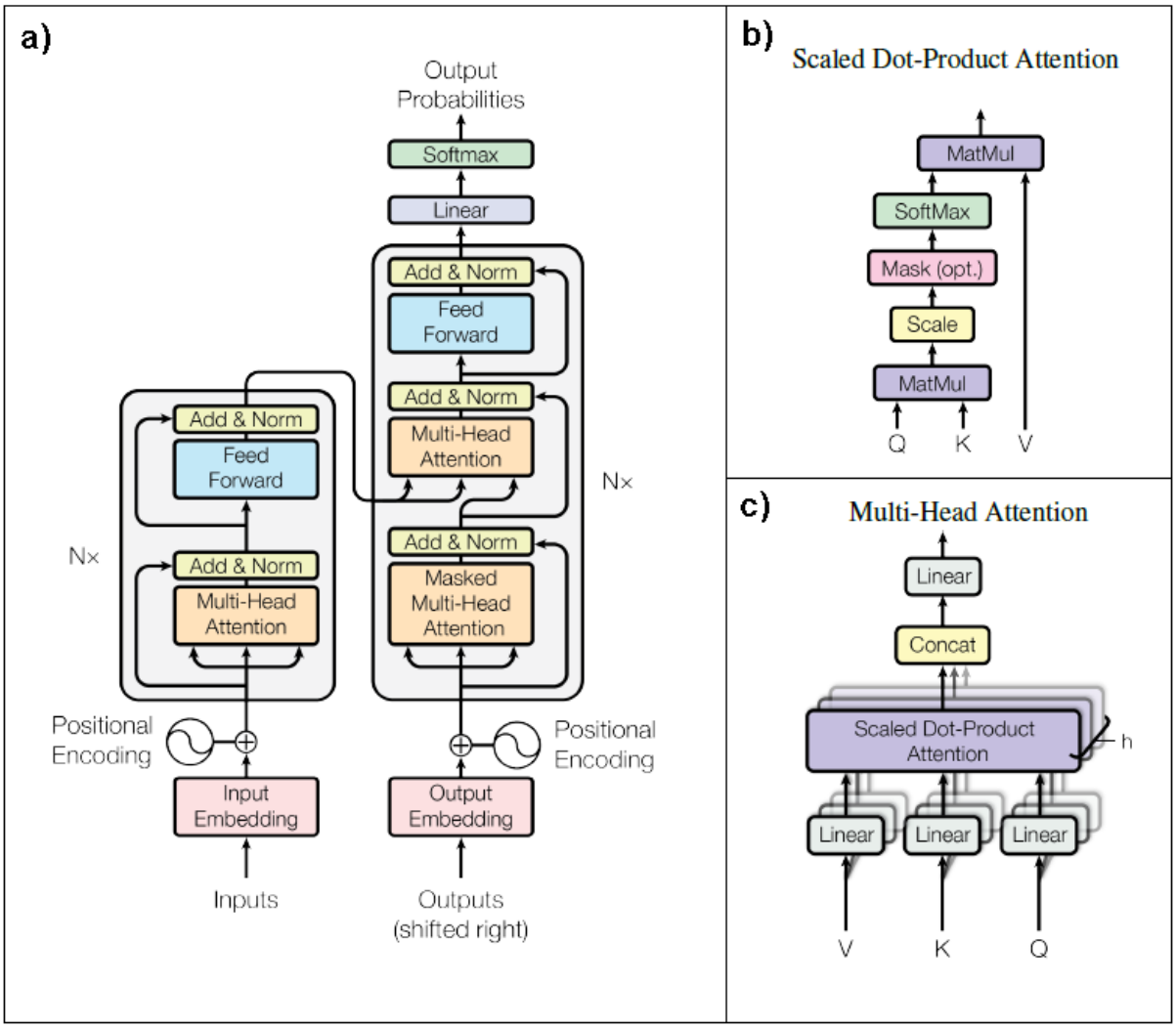


Figure 2.7: (a): The original Transformer architecture. (b): Scaled Dot-Product Attention representation. (c): Multi-Head Attention representation (Images from [42]).

The input information flows through the following model components:

1. **Positional encoding.** The attention mechanism, as will be described later, does not take into account the absolute and relative position of input elements: a mechanism to keep into account the sequential information is thus needed. This is provided by the positional encoding layers: they sum to each vector embedding of the input ordered values of a periodic function F. If d_{model} is the embedding dimension, the

contribution to each input elements is:

$$PE_{(pos,i)} = F\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{2.8}$$

where pos is the position and i is the dimension of the input elements. Multiple choices are possible for the periodic function F; in the original implementation, a sine/cosine approach is presented:

$$F(i,x) = \begin{cases} \sin(x), & i = 2k \\ \cos(x), & i = 2k+1 \end{cases} \tag{2.9}$$

thus resulting in the following:

$$PE_{(pos,2k)} = \sin\left(\frac{pos}{10000^{\frac{2k}{d_{model}}}}\right) \tag{2.10}$$

$$PE_{(pos,2k+1)} = \cos\left(\frac{pos}{10000^{\frac{2k}{d_{model}}}}\right) \tag{2.11}$$

in this way, each input dimension is associated to a sinusoid; its frequency varies with the element's position. Why is this mechanism able to encode relative positions? Consider the rate of change of bits in binary numbers, as depicted in Fig.2.8. The changing frequency, from right to left, of the first bit is $\frac{1}{2}$, of the second is $\frac{1}{4}$, of the third is $\frac{1}{8}$ and so on. To each bit position is associated a certain frequency; the sine/cosine encoding represents the float continuous counterpart of this mechanism [18]. Furthermore, by dividing two frequencies it is possible to obtain the relative distance between their two associated positions.

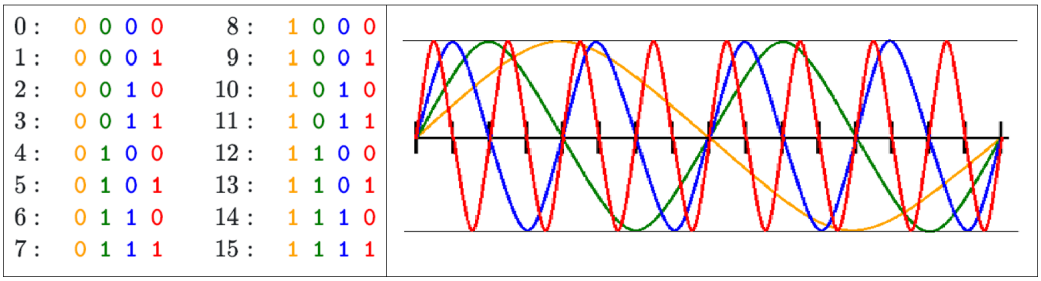


Figure 2.8: Representation of the sine/cosine encoding (Left image from [1]).

2. **Encoder and decoder stacks.**

   The encoder and decoder blocks of the architecture are composed by a stack of N_{enc} and N_{dec} identical blocks (6 and 6 in the original implementation). Each block contains two layers: a *multi-head self-attention* layer, where the self-attention mechanism takes place, followed by a *feed-forward* layer. Between each layer is also performed a residual connection, in order to preserve a part of the pre-layer input, and a layer normalization is then applied to the result.

   With respect to the encoder, the decoder presents two differences. The first is that the attention performed by the first decoder block is *masked*, in order to prevent input elements from attending to future outputs: the predictions for each position t can depend only on the known outputs at positions less than t. The second resides in the fact that the decoder blocks provide a third multi-head self-attention sub-layer, in order to collect the output of the encoder stack.

3. **Linear layer with softmax function.**

   After the decoder, a final linear layer, followed by a softmax activation function, is in charge of providing the final output. The layer size d_l depends on the application: for classification tasks, it corresponds to the number of classes; for translation tasks, it is equal to the vocabulary size. When the output is an array of floats (such as in the TSF case),

multiple adaptions can be made: one of them involves removing the softmax activation and taking d_l equal to the forecasting window.

The self-attention mechanism

A visualization of the multi-head attention performed in the encoder and decoder stacks is depicted in Fig.2.7b and Fig.2.7c. In summary, the mechanism consists in a linear layer taking as input a concatenation of h different *scaled dot-product attentions*, each performed by a different *attention head*. For each head, three linear layers are in charge of extracting from the input a tercet (Q, K, V) of *queries*, *keys* (both of dimension d_k) and *values* (of dimension d_v), upon which the attention is computed. The attention is computed as follows:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.12}$$

where the scaling constant $\frac{1}{\sqrt{d_k}}$ is added to prevent dot products from growing too large in magnitude and thus hindering the softmax activation.

Aim of the self-attention is to relate different positions of a single sequence in order to compute a meaningful representation of it; in order to do so, the self-attention stores into a matrix a *compatibility score* of each possible query-key combination, and uses these scores to compute a weighted sum of the values. The rationale behind it is that values associated to a higher query-key score are considered as ”more meaningful” in terms of information, and thus should contribute more to the final output representation. An example of self-attention matrix is depicted in Fig.2.9.

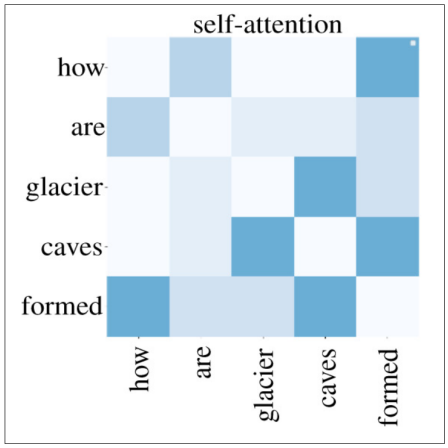


Figure 2.9: Example of query-key scores on their corresponding self-attention matrix (Image from [12]).

Why using multiple attention heads? Each head comes with its own parameters, and thus during the training different heads can focus on different parts and internal dependencies of the input. So, each head is associated to a different semantic information, and the concatenation of their output allows for a greater extraction and retain of useful information.

Chapter 3

Related Work

3.1 Transformer drawbacks and state of the research

Despite the effectiveness of the *Transformer* model, some studies [11][24] suggested some weaknesses in the original architecture, and depending on the application many enhancement proposals have been made.

Regarding the TSF problem, the work of Li et al. [11] shows that the vanilla Transformer is *locally agnostic*: the attention mechanism matches queries and keys without taking in consideration their local context (namely, neighbouring elements in the input sequence), thus being prone to anomalies and mislead by outliers.

Another weakness resides in the *positional encoding*: since the attention mechanism does not explicitly take into account sequentiality, this knowledge must be injected to the input through the positional encoding, at the risk of a loss in meaningful information. Furthermore, while the sine/cosine encoding is able to capture the information about both the absolute and the relative position, no notion of *time* is involved: the order in which two elements occur is taken into account, but their temporal distance is not.

A third, critical aspect of the Transformer's attention lies in its *computational complexity*: given a sequence length L, the time and memory burden is $O(L^2)$, making it difficult to learn patterns in long series [24].

These three represent the major points of weakness of the original model, starting from which various transformer-based architectures have been proposed in the literature and many improvements have been made. Other models focus instead on the problem of interpretability [25] and the use of Transformers with an unsupervised approach [46].

It is worth noting that the majority of these proposals derive from the enhancement of one or both of the vanilla *Transformer*'s two main features: the *positional encoding* and the *attention mechanism*. The following paragraphs will present some of the most recent architectures providing an improvement with respect to the aforementioned problems.

3.2 Models focusing on local context of input

It is common for time series to encounter at some time steps certain salient events, that depending on the application can be seen as *anomalies*, impactful enough to determine a shift in the pattern of subsequent values. To provide an example, the advent of a blackout would cause a sheer drop in a series monitoring the electrical consumption of the city it takes place in. Consequently, the information is locally sensitive: series elements with equal values provide different insight if one of them is temporally near an anomaly while the other is not.

Since the vanilla Transformer does not take into account this kind of information, the work of Li et al. [24] proposes the introduction of *causal convolutions* into the attention mechanism: this method, depicted in Fig.3.1, involves the use of convolution kernels to construct queries and keys, and is carried out by considering only the past neighbours of each input element. In this way, the local context of single entries is involved in the subsequent attention

operation.

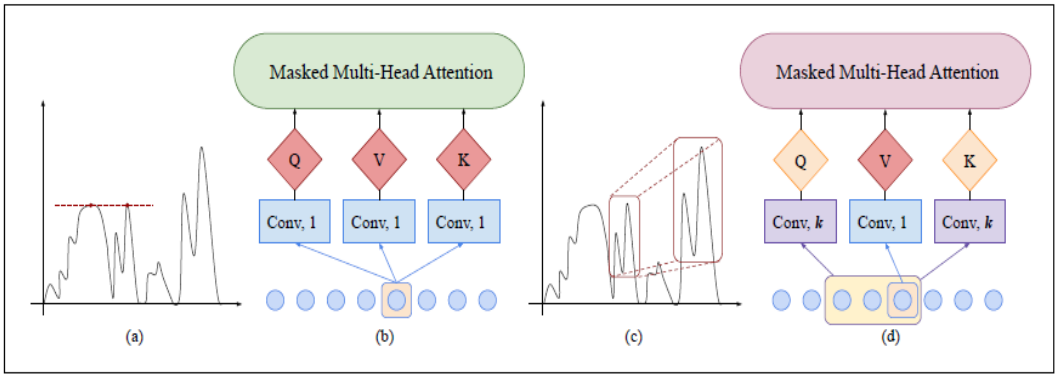


Figure 3.1: Comparison between the classical query-key construction (b) and the *causal convolution* one (d), and the portion of input they involve (a, d). The first method is locally-agnostic, while the second one is context-aware (Image from [24]).

A second approach to the local context problem in TSF is provided by the *SpringNet* architecture, by Koprinska et al. [28]. The authors, citing the Li et al. work [24], underlined a limitation on the use of causal convolutions to capture local information in time series, due to the fact that after each convolution the input sequence is projected into a lower-dimensional latent space and thus the local shape of the series is distorted. For this reason, they proposed the use of the Spring algorithm (Sakurai et al. [35]), which is able to find subsequences in data streams that are similar to a query one by means of the *Dynamic Time Warping* (DTW) trajectory similarity measure. Given two input sequences, the DTW is able to determine their affinity while being robust with respect to temporal distortions such as shifts and scalings.

In the *SpringNet* model, the DTW is used as a distance measure on the *SpringDTW Attention Layers*, on which the Spring algorithm identifies the subsequences of keys that match query series. This mechanism allows the architecture to be effective in TSF applications involving recurrent anomalies, responsible for local fluctuations in the series.

3.3 Models with focus on efficiency

Since the original Transformer release, a plethora of methods have been suggested in the literature to improve the computational and memory efficiency of the vanilla attention mechanism.

Li et al.[24] proposed, in their *LogSparse Transformer*, the use of *LogSparse attention*, reducing the complexity of attention computation from $O(L^2)$ to $O(L(logL)^2)$ while maintaining high performances. The rationale behind the *LogSparse attention* comes from the assumption that taking the full input sequence for the attention mechanism is redundant and comes with a computational cost that could be reduced. Thus, in each LogSparse layer a *sampling* of input elements is made, by following an exponential step size: by considering a base of 2, at each time step t only the elements $\{t, t-1, t-2, t-4, t-8, ...\}$ are taken. It is also worth to notice that by using an exponential sampling step the majority of samples is near to the current time step, following the idea of importance of the local context. A comparison between the Full and LogSparse attention methods is depicted in Fig.3.2.

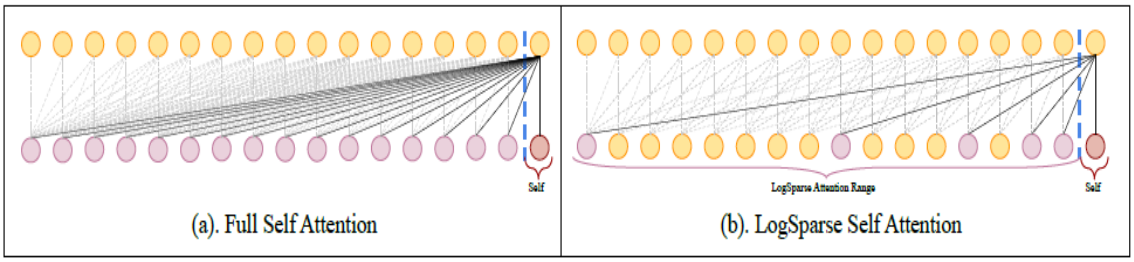


Figure 3.2: Comparison between the vanilla attention (a) and the LogSparse attention (b). (Image from [24]).

The idea of considering a subset of the input on attention layers in order to save computational power is taken up by many other models. But while most of them approximate the attention matrix by applying some notion of *sparsity* to it, the methods to do so may vary significantly. The work of Tay et al. [39] proposes a survey of the main models of this category, which can be divided

by the approximation technique:

- **Pattern approximation.** This simple method consists in taking a subsample of the input by following determined fixed patterns, which can be *blockwise* if considering windows of a fixed length (*Blockwise Transformer* [33], *Local Transformer* [31]), *strided* if attending at fixed intervals (*Sparse Transformer* [5], *Longformer* [3]) or *compressed* if the input sequence is down-sampled by means of a pooling operator (*Compressed Attention Transformer* [29]). It is also possible to blend two or more of these distinct patterns, resulting in a *combination of patterns* approximation (*Axial Transformer* [14]), or connect multiple blocks by means of *recurrence* (*Transformer-XL* [8]).

- **Learnable patterns approximation.** This technique extends the previous one by considering the pattern choice as part of the training process. Models falling in this sub-category typically make use of a similarity measure to sort input tokens (*Sparse Sinkhorn Attention Transformer* [38]) or divide them into clusters (*Reformer* [19], *Routing Transformer* [34]).

- **Memory methods.** This approach involves the training of a *side memory* to compress the input sequence and store temporary context information that will be used as a shortcut for future processing (*Set Transformer* [22]).

- **Low-rank and kernel methods.** These methods are finalized to avoid explicitly computing the full query-key attention matrix, either by a projection to a lower-dimensional representation (*Linformer* [43]) or an approximation of the attention mechanism through the application of kernels (*Linearly Scalable Long-Context Transformer* [6]).

It is important to underline that these techniques are not mutually exclusive, and a single model can make use of a combination of them. The full list

of efficient models surveyed by Tay et al., along with their classification and the computational complexity of their attention layers, is provided in Tab.3.1.

Model	Complexity	Class
Memory Compressed (Liu et al., 2018)	$O(n_c^2)$	P+M
Image Transformer (Parmar et al., 2018)	$O(n \cdot m)$	P
Set Transformer (Lee et al., 2019)	$O(n \cdot k)$	M
Transformer-XL (Dai et al., 2019)	$O(n^2)$	RC
Sparse Transformer (Child et al., 2019)	$O(n\sqrt{n})$	P
Reformer (Kitaev et al., 2020)	$O(nlog(n))$	LP
Routing Transformer (Roy et al., 2020)	$O(nlog(n))$	LP
Axial Transformer (Ho et al., 2019)	$O(n\sqrt{n})$	P
Compressive Transformer (Rae et al., 2020)	$O(n^2)$	RC
Sinkhorn Transformer (Tay et al., 2020)	$O(b^2)$	LP
Longformer (Beltagy et al., 2020)	$O(n(k+m))$	P+M
ETC (Ainslie et al., 2020)	$O(n_m^2 + n \cdot n_m)$	P+M
Synthesizer (Tay et al., 2020)	$O(n^2)$	LR+LP
Performer (Choromanski et al., 2020)	$O(n)$	KR
Linformer (Wang et al., 2020)	$O(n)$	LR
Linear Transformers (Katharopoulos et al., 2020)	$O(n)$	KR
Big Bird (Zaheer et al., 2020)	$O(n)$	P+M

Table 3.1: Efficient transformer models surveyed by Tay et al., along with their attention mechanism complexity and their classification. Complexity abbreviations: n = sequence length, $\{b, k, m\}$ = pattern window/block size, n_m = memory length, n_c = convolutionally compressed sequence length. Class abbreviations: P = Pattern, M = Memory, LP = Learnable Pattern, LR = Low Rank, KR = Kernel, RC = Recurrence. (Original table from [39]).

3.4 Models with focus on positional and temporal information

The way the attention mechanism works restricts the transformer model from fully exploiting the sequential nature of the input, and the positional encoding only partially makes up for it. In order to tackle this problem, many alternative approaches can be found in the literature.

The work of Shaw et al. [37] presents an efficient way of incorporating relative position representations in the self-attention computation. The proposed *Relation-aware Self-Attention* treats the input as a labeled, directed, fully-connected graph, which edges capture information about the relative position differences between input elements. The edge information is extracted and exploited both in the query-key compatibility computation and as a final contribution to the attention sublayer output, allowing for a position-aware version of self-attention. This idea is further enhanced and optimized by Huang et al.'s *Music Transformer* [15], in which the *relation-aware attention* is implemented in an efficient way by means of a "skewing" algorithm while maintaining its peculiar properties.

Another interesting approach is provided by Fan et al. in their proposed *Feedback Transformer* [11]. The novelty of this model resides in the use of a *global memory*, accessible by all layers, which takes part in the computation and whose content is updated at each time step with an embedding of the layers hidden states. This mechanism feeds past elaborations into future time steps, allowing the model to compute and transform inputs in a recursive way, similarly to how a RNN works. The working principle of the Feedback Transformer and a comparison with the vanilla transformer are depicted in Fig.3.3.

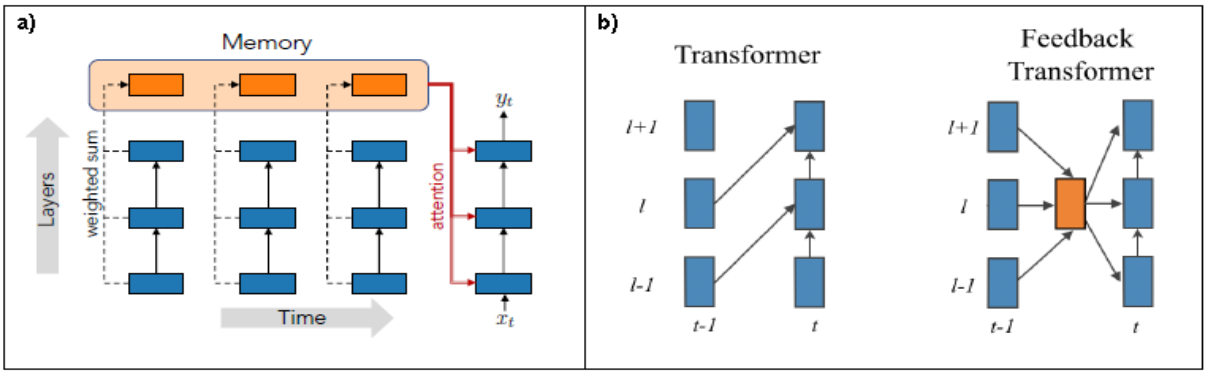


Figure 3.3: (a): Working principle of the Feedback Transformer: past hidden representations from all layers are merged into a single vector and stored in a global memory. (b): Comparison between vanilla and Feedback transformer architectures. (Image from [11]).

A different technique to augment the sequential information considered by transformer-based models consists in substituting the positional encoding with different methods to capture the ordering of the data. This is the case of the *seq2tens* encoding, proposed by Toth et al. [40]: the non-commutativity of the input sequence is captured by first associating abstract features to each input element by means of a *static feature map*, and by subsequently merge these features together in a larger vector space. While the authors do not explicitly consider transformer-based models in their dissertation (their focus is on enhancing CNNs and RNNs), *seq2tens* could be easily used as a replacement for the vanilla positional encoding.

In a similar manner, the *time2vec* encoding, proposed by Kazemi et al. [17], could be adopted. This method translates the notion of sequentiality into the one of *time*, and can be seen as the extension of the positional encoding from a discrete synchronously-sampled time to the continuous. This could prove invaluable when working with time series, and therefore in TSF problems, since the input order is as important as the time distance between elements.

For a given scalar notion of time t, the time2vec of t, in notation $t2v(t)$, is

a vector of size $k + 1$ defined as follows:

$$t2v(t)[i] = \begin{cases} w_i t + \phi_i & \text{if } i = 0 \\ f(w_i t + \phi_i) & \text{if } 1 \leq i \leq k \end{cases} \tag{3.1}$$

where $t2v(t)[i]$ is the i^{th} element of $t2v(t)$, f is a periodic activation function (such as sine/cosine), and $w_i t, \phi_i$ are learnable parameters, while the encoding size $k + 1$ is added as a model hyperparameter. With respect to the positional encoding, *Time2vec* comes with some nice properties:

- It is **model-agnostic**. Due to its simplicity, *Time2vec* can be easily imported into different architectures and improve their performances, without compatibility issues.
- It is **invariant to time rescaling**. Given an arbitrary scale factor α, so that each time step t is mapped into αt, it suffices to similarly scale each parameter w_i to αw_i in order to be applied to the scaled data.
- It can **capture both periodic and non-periodic patterns**. Working ith time series, the linear term (for $i = 0$) and the periodic one (for $1 \leq i \leq k$) allow to address separately these two components.

Overall, *time2vec* encoding represents a strong tool to approach problems in which time is an important feature; this is the case when dealing with time series and therefore in TSF problems.

3.5 Other transformer-based models

Aside from the aforementioned problems, some proposed works in the literature aim at enhancing transformer-based models with respect to typical issues shared by the majority of deep learning architectures.

A major research topic is about *explainability*: most of the state-of-the-art models are still used as *black boxes*, on which it remains difficult to determine which aspects of the provided input drive the output decisions. This is

surely true for simpler models, such as CNNs and RNNs; for transformers, the attention mechanism represents a first step towards explainability, but in most applications the underlying decision process still remains obscure. In this context, Lim et al. proposed the *Temporal Fusion Transformer (TFT)* [25], a multi-horizon forecasting architecture which also provides insight into how and which parts of the input are considered in order to make predictions. The TFT structure, depicted in Fig.3.4, is constituted by five key components:

- *Variable selection networks*, to select relevant input variables at each time step;
- *Gating mechanisms*, to skip over any unused components of the architecture (which may vary depending on the application);
- *Static covariate encoders*, to integrate static features into the network;
- *Sequence-to-sequence layers*, to take into account local short-term temporal relationships;
- *interpretable multi-head attention blocks*, to capture long-term dependencies while enhancing their output explainability.

Furthermore, the output comes in the form of *prediction intervals*, to determine the confidence range of target values at each prediction horizon.

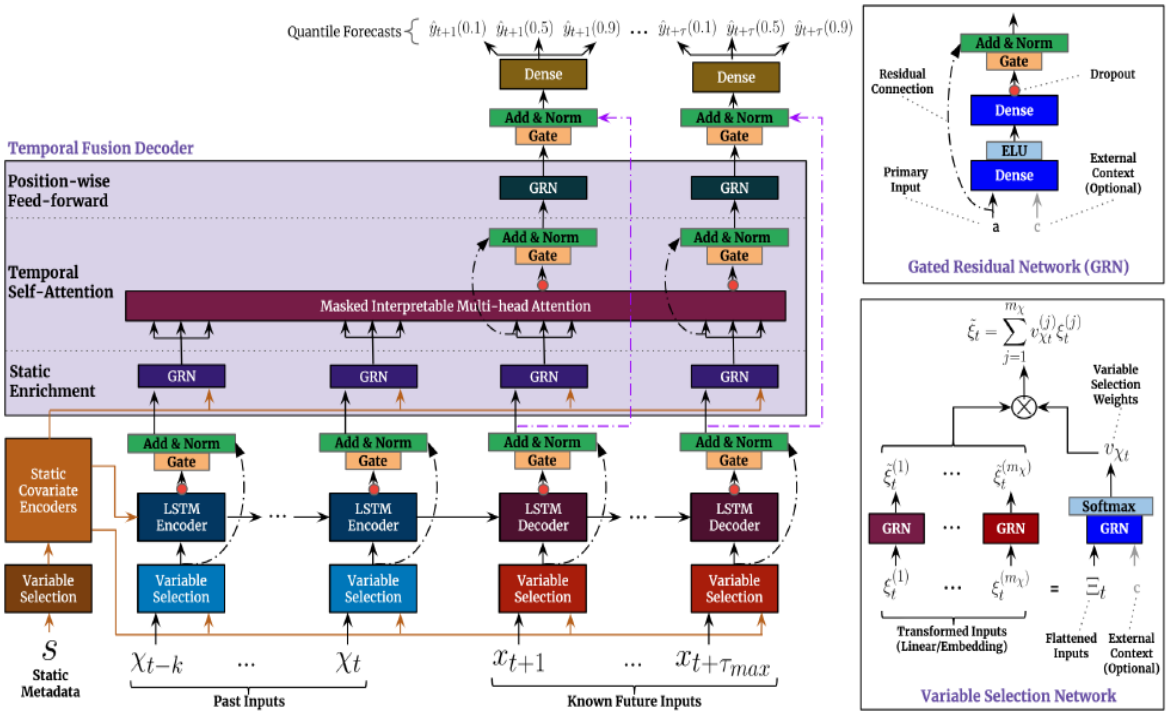


Figure 3.4: Temporal Fusion Transformer model architecture. (Image from [25]).

Another interesting proposal is the *Transformer-based framework for multivariate time series representation learning*, by Zerveas et al. [46]. The novelty of this proposal resides in the fact that the framework includes an unsupervised pre-training scheme which is able to work with unlabeled time series data. The authors show how this pre-training proves beneficial for applications such as regression and classification of time series, even if the model is trained with a limited number of training samples both in the unsupervised and the supervised steps.

Chapter 4

The Datasets

In order to employ the models of this thesis, two datasets have been chosen: *ETT Dataset* [47] and *CU-BEMS* [32]. While being substantially different, they can be linked to two practical time series forecasting problems, each with their own challenges. The following paragraphs will provide a description of the data they enclose, along with the practical scenarios correlated to them.

4.1 The ETT dataset

The *Electricity Transformer Temperature* (ETT) dataset [47]

It contains a multivariate time series regarding electrical transformer oil data coming from two different stations located in separate countries of China. For each station, both the 15-minutes and 1-hour timestep versions are available, thus resulting in four sub-datasets: *ETTh1*, *ETTh2*, *ETTm1* and *ETTm2*. Each of their data point consists of 8 features: the time step, the predictive value ”oil temperature”, and 6 different types of external power load features, as depicted in Tab.4.1.

Feature	Meaning
date	Date and time of the sample
HUFL	High UseFul Load
HULL	High UseLess Load
MUFL	Medium UseFul Load
MULL	Medium UseLess Load
LUFL	Low UseFul Load
LULL	Low UseLess Load
OT	Oil Temperature

Table 4.1: Features of data points in the four ETT datasets.

As will be shown in subsequent chapters, the ETTm1 dataset has been chosen between the four in order to train the models of this thesis work and evaluate their performances. As shown in Fig.4.1, it is comprised of $69'679$ elements, covering measurements between $01/07/2016$ and $26/06/2018$, almost two years of data. A plot of the target "Oil Temperature" variable on the entire dataset and some zoomed windows at monthly, weekly and daily size are depicted in Fig.4.2.

	date	HUFL	HULL	MUFL	MULL	LUFL	LULL	OT
0	2016-07-01 00:00:00	5.827	2.009	1.599	0.462	4.203	1.340	30.531000
1	2016-07-01 00:15:00	5.760	2.076	1.492	0.426	4.264	1.401	30.459999
2	2016-07-01 00:30:00	5.760	1.942	1.492	0.391	4.234	1.310	30.038000
3	2016-07-01 00:45:00	5.760	1.942	1.492	0.426	4.234	1.310	27.013000
4	2016-07-01 01:00:00	5.693	2.076	1.492	0.426	4.142	1.371	27.787001
...	...	...	...	...	...	...	...	...
69675	2018-06-26 18:45:00	9.310	3.550	5.437	1.670	3.868	1.462	9.567000
69676	2018-06-26 19:00:00	10.114	3.550	6.183	1.564	3.716	1.462	9.567000
69677	2018-06-26 19:15:00	10.784	3.349	7.000	1.635	3.746	1.432	9.426000
69678	2018-06-26 19:30:00	11.655	3.617	7.533	1.706	4.173	1.523	9.426000
69679	2018-06-26 19:45:00	12.994	3.818	8.244	1.777	4.721	1.523	9.778000

Figure 4.1: Head and tail of the ETTm1 dataset.

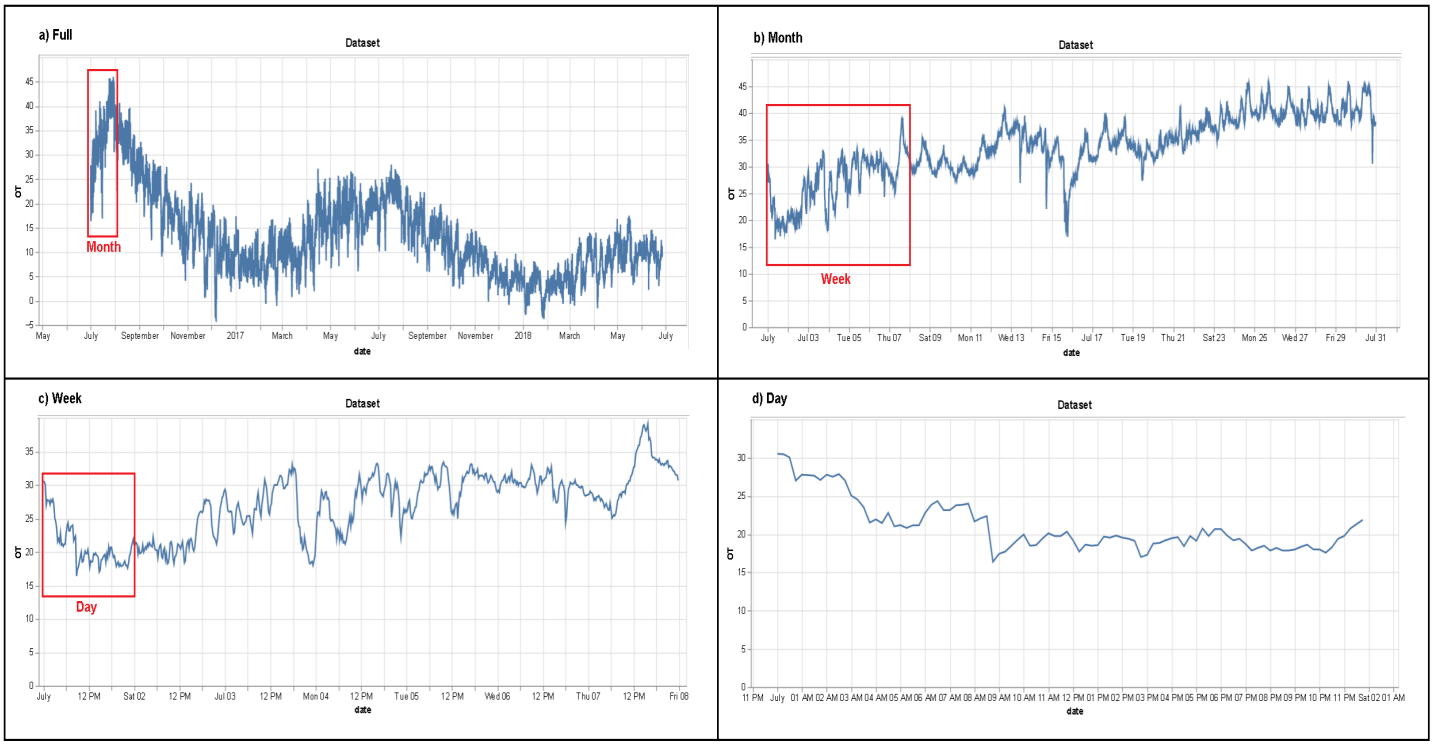


Figure 4.2: Plot of the full ETTm1 dataset (a) and zoomed windows of monthly (b), weekly (c) and daily (d) sizes.

By looking at the target variable shape, some considerations can be made. First of all, it is possible to observe a yearly-long *seasonal pattern*: the oil reaches its maximum temperature during the summer months (july-august) and has its minimum in the winter ones (december-january). Still, the peaks are different for each year, and aside from this no other significant pattern can be seen. At monthly and weekly levels, the series shows irregular fluctuations, while at daily level there is some short-term local continuity, slightly distorted by noise.

As for the ”power load” auxiliary variables, the situation is different: by looking at their autocorrelation plot depicted in Fig.4.3, strong daily patterns can be observed.

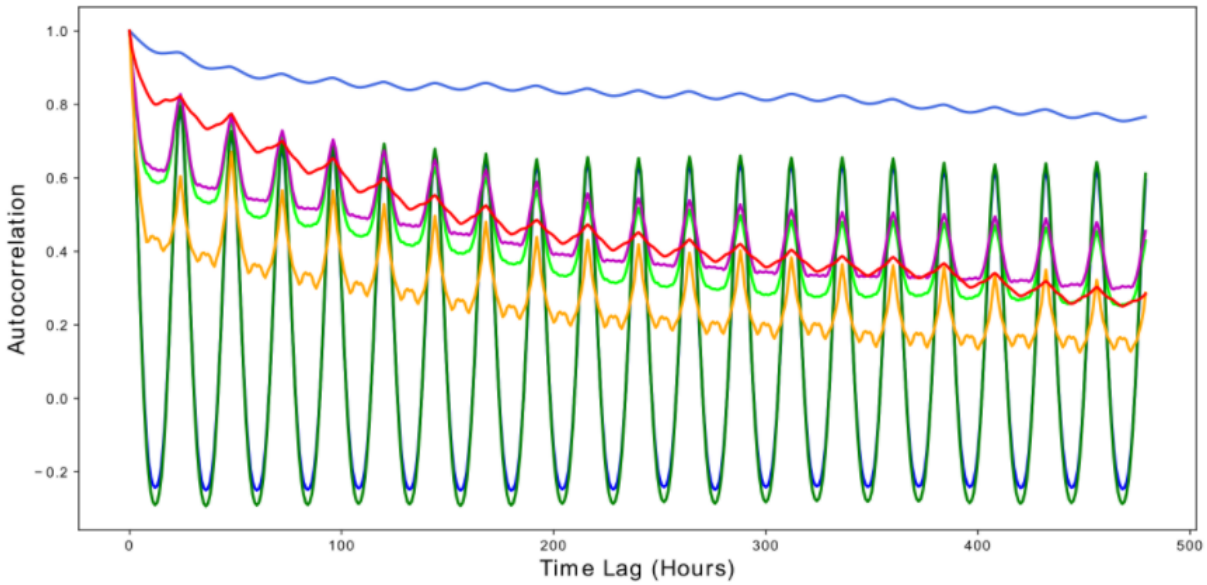


Figure 4.3: Autocorrelation graph of the ”Oil Temperature” target variable (upper blue line) and the six auxiliary ”Power Load” covariates (lower lines). While the first shows some degree of local continuity, the latter shows short-term daily pattern (every 24 hours) and long-term week pattern (every 7 days) (Image from [47]).

Overall, predicting future oil values of the *ETTm1* dataset represents a

challenging TSF problem, due to the short-term irregularities of the target series. But a correct forecast could bring strong benefits: as described in [47], anticipating the electric power demand of specific areas is problematic due to its variation with respect to factors such as weekdays, holidays, seasons, weather and temperatures. For this reason, reliable methods to perform long-term predictions of the demand itself with an acceptable precision still do not exist, and a wrong prediction could overheat the electrical transformer, damaging it. Since the oil temperature can reflect the condition of the electrical transformer, its prediction could be used in order to employ an *anomaly detection* mechanism: by comparing the expected behaviour with the currently measured one, if their difference falls over a certain threshold an alarm signal will be sent, and appropriate actions could be taken if deemed necessary. Moreover, since the oil temperature is related to the actual power usage, an indirect estimation of the latter could be obtained, preventing overestimations and thus unnecessary wastes of electric energy and equipment degradation.

4.2 The CU-BEMS dataset

The *Chulalongkorn University Building Energy Management System* dataset, or CU-BEMS [32], is an extensive collection of data comprising electricity consumption and indoor environmental measurements of a seven-story $11,700m^2$ office building located in Bangkok, Thailand (Fig.4.4).

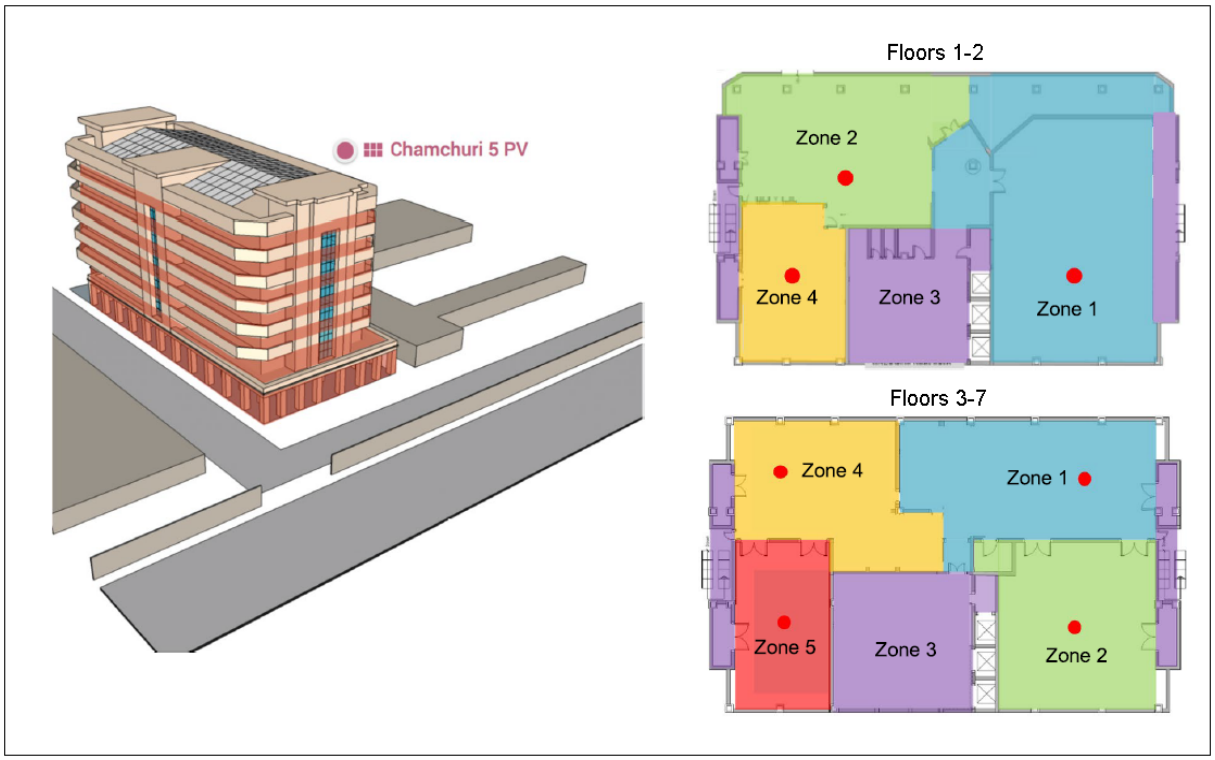


Figure 4.4: Visualization of the cubems-related seven-story office building (a) and floor planimetry (b) (Image from [32]).

Each floor of the building is divided into four (for floors 1-2) or five (for floors 3-7) zones, and each zone is subjected to six different measurements:

- Electrical consumption of air conditioning units (AC);
- Lighting load;
- Plug load;
- Indoor temperature;
- Relative humidity;
- Ambient light.

The data is available at one-minute granularity, and covers 1,5 years of measurements, from $01/07/2018$ to $01/01/2020$. While some missing values

are present, the majority of features have a data availability of at least 95%, with some exceptions at middle floors. Being divided both by year and by floor, CU-BEMS is composed by 14 sub-datasets; a summary of the overall structure is depicted in Fig.4.5.

a)

	Year 2018	Year 2019
Floor1	*2018Floor1.csv*	*2019Floor1.csv*
Floor2	*2018Floor2.csv*	*2019Floor2.csv*
Floor3	*2018Floor3.csv*	*2019Floor3.csv*
Floor4	*2018Floor4.csv*	*2019Floor4.csv*
Floor5	*2018Floor5.csv*	*2019Floor5.csv*
Floor6	*2018Floor6.csv*	*2019Floor6.csv*
Floor7	*2018Floor7.csv*	*2019Floor7.csv*

b)

File name	Category	Measurement (time-series)	Unit
YFloorX.csv	Electricity consumption data	Individual air conditioning (AC) unit	kW
for X ∈ [1, …, 7]		Lighting load	kW
		Plug load	kW
for Y ∈ [2018, 2019]	Indoor environmental sensor data	Indoor temperature	°C
		Relative humidity	%
		Ambient light	lux

c)

File name	Zone No.	AC	Light	Plug	Sensor	No of Data Columns
Floor7.csv	Zone 1	4	1	1	3	29
	Zone 2	1	1	1	3	
	Zone 3	0	1	1	0	
	Zone 4	1	1	1	3	
	Zone 5	1	1	1	3	

Figure 4.5: CU-BEMS dataset file names (a), types of available measurements (b) and classification of features contained in the dataset of floor 7 (c) (Original images from [32]).

For the purposes of this thesis, the original CU-BEMS data has undergone some preliminary processing steps. First of all, it has been decided to work at floor-level, only considering data from floor 7 as the context of predictions. The seventh one in particular has been chosen for two main reasons: it is one of the floors with the most number of sensors in it, leading to 29 corresponding features (Fig.4.5c), while at the same time containing the least amount of missing values.

Secondly, a 15-min downsampling of the data has been carried out: this has been done not only to adopt the same sample frequency of ETTm1, but also because in the considered forecasting problem a 1-minute granularity has been deemed redundant and computationally inefficient (more input elements to compute, without a real gain in meaningful information).

At last, the object of forecasting had to be defined; concerning this, the *total floor consumption* has been computed and inserted in the dataset as the target feature. Its values, at each time step, are given by the sum of all the AC, light and plug electricity consumption in the floor, regardless of the zone; a plot of this constructed series is depicted in Fig.4.6.

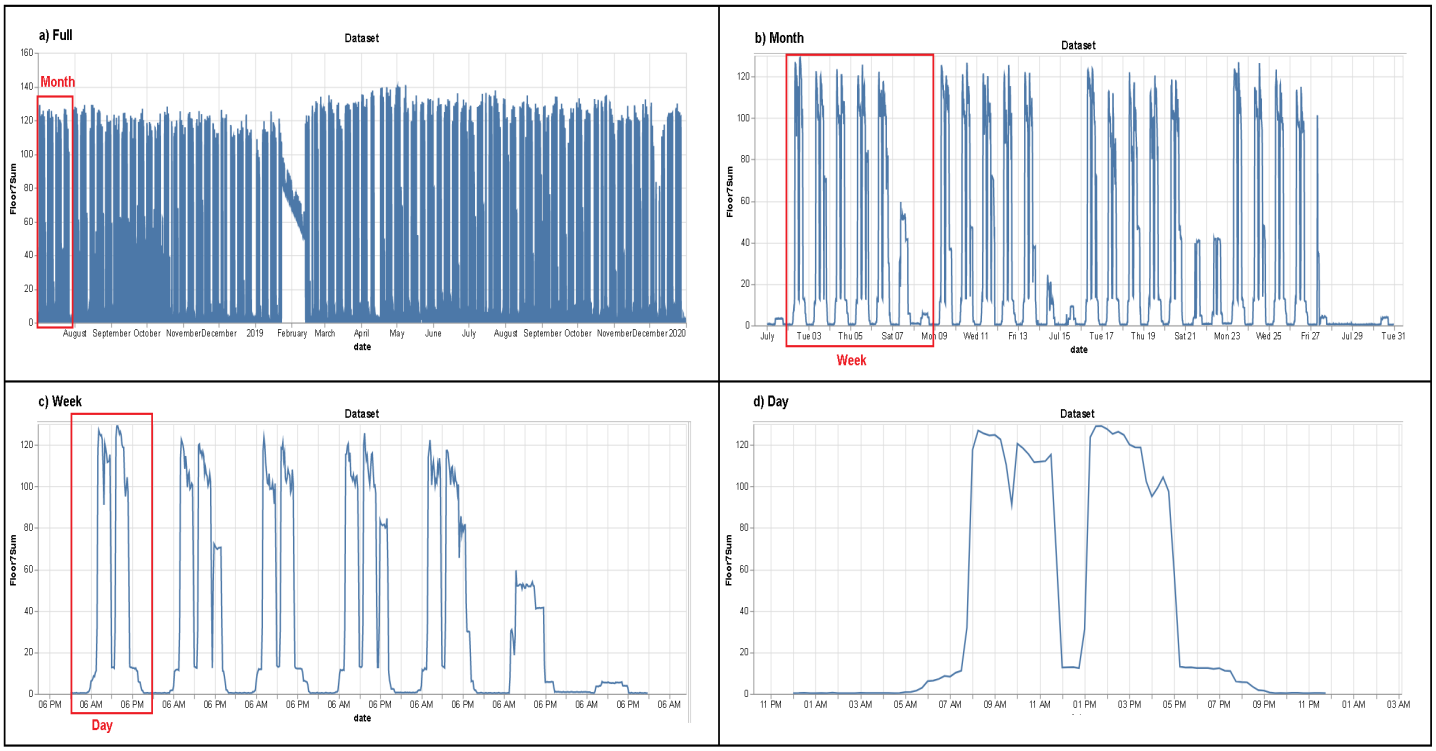


Figure 4.6: Plot of the 15-minutes sampled ”Total Floor 7 Consumption” feature inserted in the CU-BEMS dataset (a) and zoomed windows of monthly (b), weekly (c) and daily (d) sizes.

By looking at the graph, two predominant seasonality patterns can be recognized: a *daily* one (Fig.4.6d), in which the consumption peaks correspond to the typical working hours (from 8am to 5pm, with a pause at noon), and a *weekly* one (Fig.4.6c), where the highest activity is registered at working days, while weak consumption values are registered on Saturdays and almost none can be seen on Sundays. This strong regularity is broken only by *holidays*, on which the energy consumption drops to zero (due to the office building presumably being closed) regardless of the day of the week. These occurrences have been considered as *outliers*, and in order to help the models understand them an additional boolean feature, namely ”Weekend/Holiday”, has been inserted in the dataset: for each time step, the associated value is 1 if belonging

to an holiday or a weekend (Saturday or Sunday), and 0 otherwise. An example of holiday outlier is depicted in Fig.4.7.

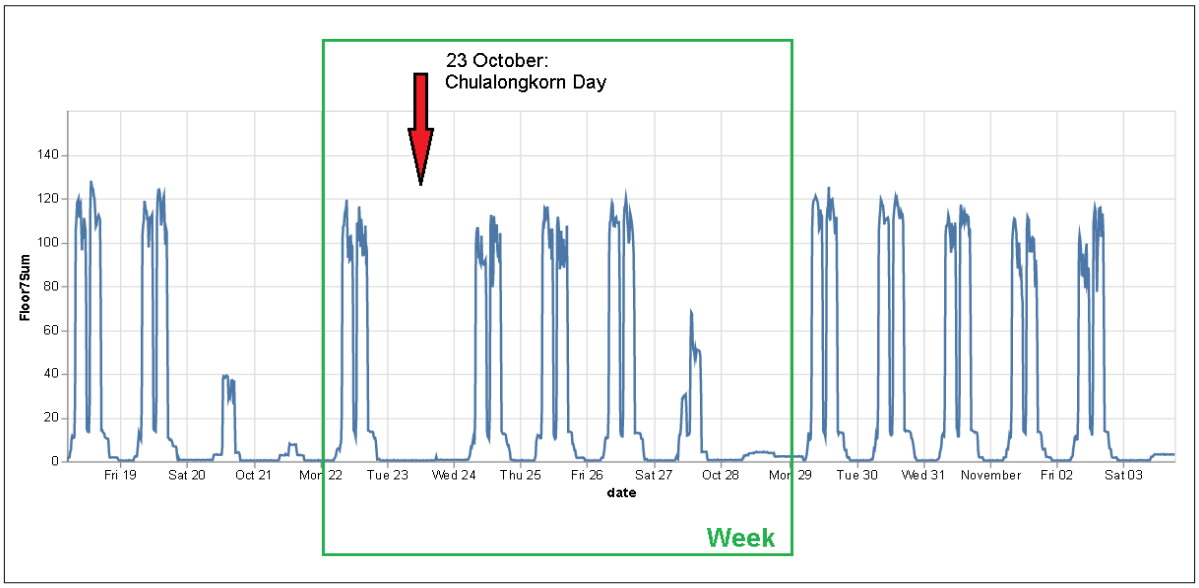


Figure 4.7: Example of daily-level outlier in the CU-BEMS dataset. Despite being a Tuesday, the 23 October date is Chulalongkorn Day, a popular holiday in Thailand, and thus the energy consumption of the building drops to zero.

As visible in Fig4.6a, in the data there is also another anomalous region, located around the period of February 2018; since this represents only the 5% of the data and differs very significantly from the rest, it has been decided to simply cut it and stitch the two remaining halves of the series by taking as the merging point the end of a week and the start of another.

The rationale behind the choice of forecasting the total energy consumption of the office building is due to the multiple possible applications it could provide. Apart from the aforementioned use of predictions as *anomaly detection* tools, the floor-level load forecasting could be used to solve *resource optimization* problems: by estimating which floors will require the most amount of electricity at a given time, it would be possible to optimally allocate the energy resources and thus prevent unnecessary wastes. Another valid opportunity would be the possibility to deploy and test building simulation models,

as suggested by [32].

For all of these situations, CU-BEMS data represents a valid starting point to train and test complex forecasting models. Overall, the high number of features and the strongly regular patterns of CU-BEMS makes it very different from ETTm1, despite being both related to an energy consumption context (addressed directly by the first, and indirectly by the latter). An architecture able to perform well on both would prove its ability to adapt to different situations and thus its effectiveness on important TSF applications.

Chapter 5

The Models

The experiments carried out on this thesis work are mainly focused on the study and the application in the TSF domain of two different transformer-based architectures: a *TransformerT2V* model and an *Informer* [47] model. The first one is a simple but effective adjustment of the vanilla Transformer for the time series problem, while the second is a complex architecture able to reach SOTA results. In order to compare their performances with the ones of non-transformer models, two architectures of this latter category have also been trained and evaluated on the proposed datasets: a CNN and a LSTM. The following paragraph will provide a description of these architectures, with particular attention to the *Informer* model and its main characteristics.

5.1 Convolutional and LSTM models

The proposed CNN and LSTM architectures follow a similar structure, depicted in Fig.5.1.

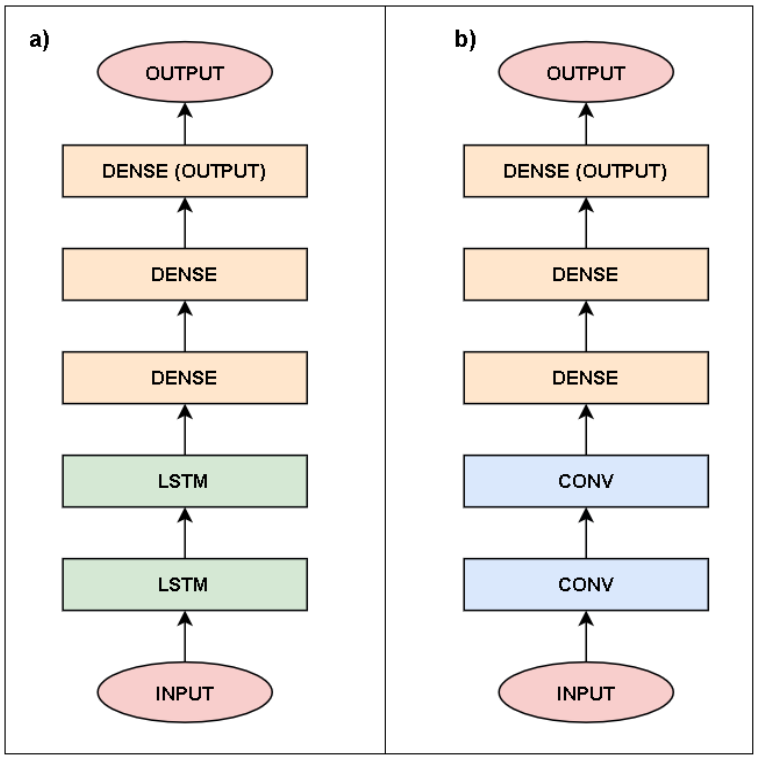


Figure 5.1: LSTM (a) and CNN (b) architectures used as representatives of non-transformer models.

Both of them are composed by two stacked CNN/LSTM layers respectively, followed by two middle dense layers and a final output one. The layers size are tunable and represent the model hyperparameters. Given the simplicity of these models and the abundance of data available for training, it has been deemed (and later confirmed by the experiments) unnecessary taking into account the problem of overfitting, and thus dropout mechanisms have not been introduced. Both models perform a multi-to-single, point estimates forecasting at a given distance in the future: given a *lookback window* L, and a *target foresight* k, if F denotes the number of features, the models take as input a tensor of size $[1, L, F]$, corresponding to the last L time steps $t - L + 1, ..., t$, and output a tensor of size $[1, 1]$, containing the predicted value of the target series at future time step $t + k$ (In the batch version, if B is the batch size, the input dimension is $[B, L, F]$ and the output one is $[B, 1]$). The choice of

letting the models focus on a single time step at a given distance in the future instead of on a target window is driven by the willingness to help the models by assigning them an easier prediction. The hyperparameters of the two models, along with their description, are listed in Tab.5.1.

Model	Hyperparameter	Description
LSTM, CNN	seq_len (L)	Length of the lookback window
LSTM, CNN	foresight (k)	Distance in the future of the predicted time step
LSTM	units_dense_lstm	Units number of the first Dense layer of the LSTM model
LSTM	units_lstm	Units number of the LSTM layers
CNN	units_dense_conv	Units number of the first Dense layer of the CNN model
CNN	filters_conv	Number of filters of convolutional layers
CNN	conv_width	Filters size of convolutional layers

Table 5.1: Hyperparameters table of the LSTM and CNN models.

5.2 The TransformerT2V model

The *TransformerT2V*, proposed as a baseline for transformer-based architectures, is depicted in Fig.5.2.

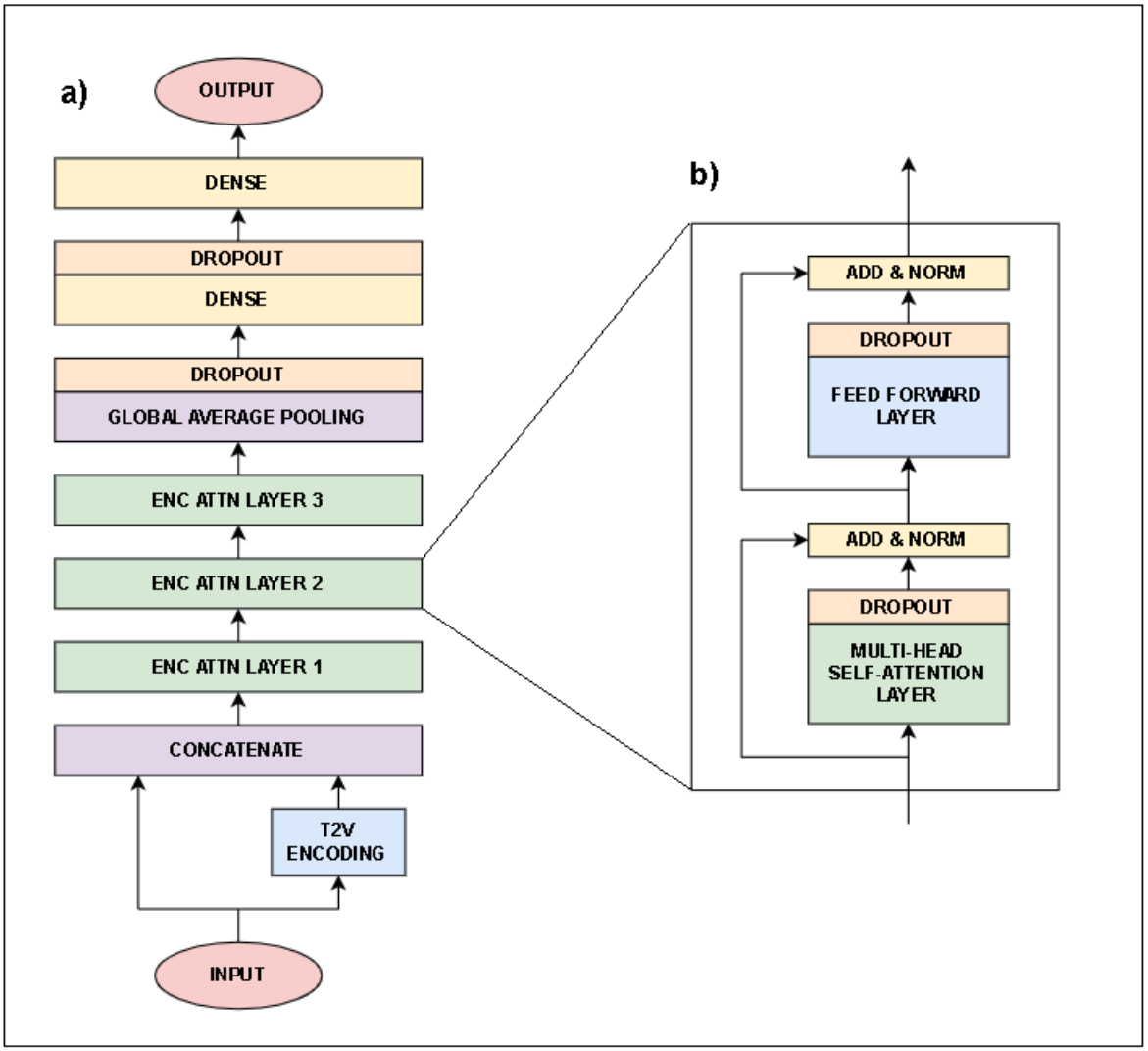


Figure 5.2: TransformerT2V architecture (a) and internal structure of the encoder attention layers (b).

Overall, the model resembles a vanilla transformer's encoder, with some slight modifications. First of all, the positional encoding is substituted by a *Time2Vec* encoding [17], in order to better take into account the temporal information of the input series. Differently from the vanilla transformer, this encoding is not added to the input, but is concatenated to it by means of an apposite block: this allows the following layers to handle the value and time components for the input separately. After the concatenation, three encoder layers apply the canonical self-attention to the processed input. The structure

of these blocks is identical to the one in the vanilla Transformer, depicted in Fig.5.2b: a multi-head self-attention layer followed by a feed-forward one, with residual connections after both of them.

The encoder stack is then followed by a *global average pooling* layer: this part of the architecture is in charge of condensing the upcoming information, reducing its dimensionality. The flattened result is then processed by two final dense layers, the last of which provides the predicted output.

All the components of the model are provided with a dropout mechanism, in order to cope with overfitting. As for the previous models, *TransformerT2V* performs a point estimates forecasting: in the batch version, given a tensor of size $[1, L, F]$, the model outputs one of size $[1, 1]$, corresponding to the predicted value at future time step $t + k$ with respect to the current time t. Also in this case, F is the number of input features, while L and k are two problem hyperparameters and represent the lookback window and the forecasting target, respectively. The complete list of *TransformerT2V* hyperparameters is depicted in Tab.5.2.

Hyperparameter	Description
seq_len (L)	Length of the lookback window
foresight (k)	Distance in the future of the predicted time step
d_{model}	Dimensionality of the representations in the Attention layers
N_heads	Number of heads in the Attention layers
FF_dim	Number of units of the Feed-Forward layers
N_dense	Number of units of the Dense layer
Dropout	Dropout rate of the model layers

Table 5.2: Hyperparameters table of the TransformerT2V model.

5.3 The Informer model

The *Informer*, proposed in 2021 by Zhou et al. [47], is a complex transformer-based architecture able to achieve state-of-the-art performances in time series forecasting applications. The model takes as input a lookback window of past timesteps in order to perform a multi-to-single forecasting, in line with the previously described architectures. However, it differs from them since the prediction is *multi-horizon*: a full window of future time steps is predicted at once.

The *Informer* structure, at a bird's eye view, is depicted in Fig.5.3 and resembles that of the vanilla Transformer, being composed by an *input embedding* mechanism, an *encoder*, a *decoder* and a final *dense layer*. However, each of these components is inherently different from its original counterpart, due to the distinct techniques they adopt to elaborate input information. In particular, the main form of novelty resides in the *ProbSparse attention*, a more efficient type of attention with respect to the canonical one. The following paragraphs will provide a description of the aforementioned *Informer* components, as well as the *ProbSparse attention* working principle.

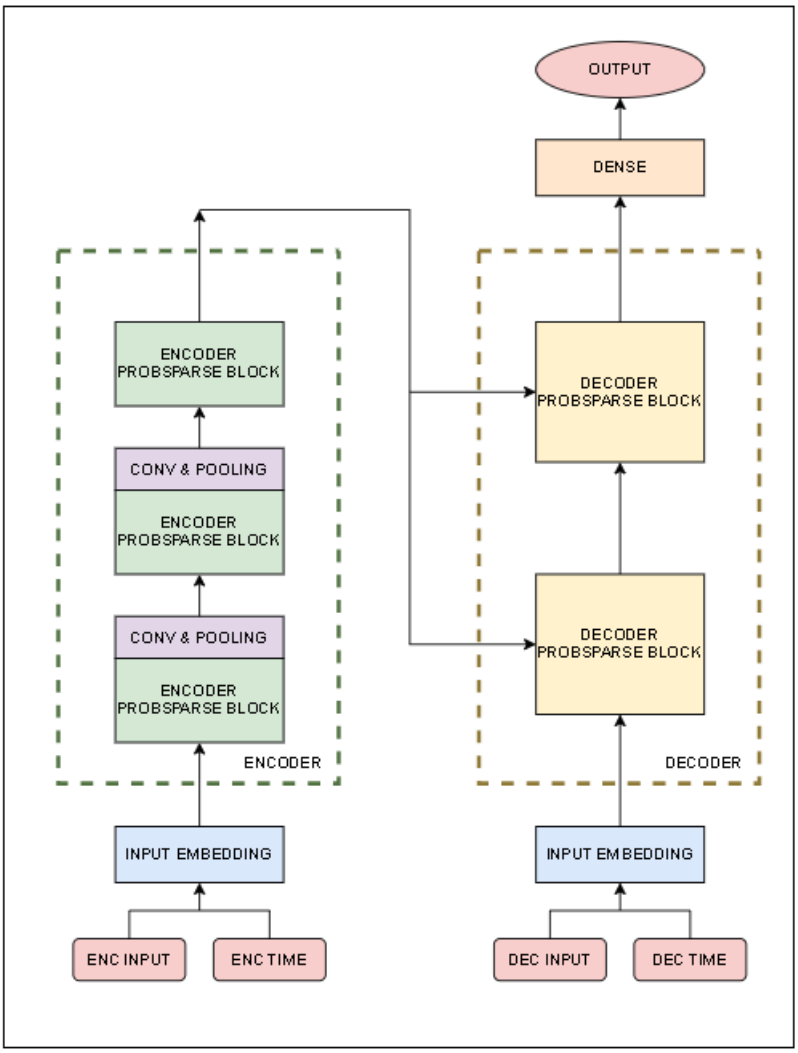


Figure 5.3: Informer architecture.

5.3.1 Starting input representation

In order to understand the *Informer*'s embedding mechanism, it is first necessary to define the target of embedding, namely the model input. The latter is divided into four parts:

- The *encoder and decoder value inputs*, two tensors corresponding to the informational content of the actual samples features. They are built, starting from a starting lookback window of size w, by following the scheme depicted in Fig.5.4. Given a *sequence length* L_s, a *label length*

L_l and a *prediction length* L_p, if F denotes the feature space dimension and the starting tensor has size $[1, L_s+L_p, F]$ (where the $t-L_s, ..., t+L_p$ time steps are covered), the resulting encoder input is of size $[1, L_s, F]$ (covering the period $t - L_s, ..., t$) and the decoder one has size $[1, L_l, F]$ (corresponding to the window $t + L_p - L_l, ..., t + L_p$). The latter is then *causally masked*, namely the values corresponding to future time steps (which represent the target of forecasting) are covered with zeros.

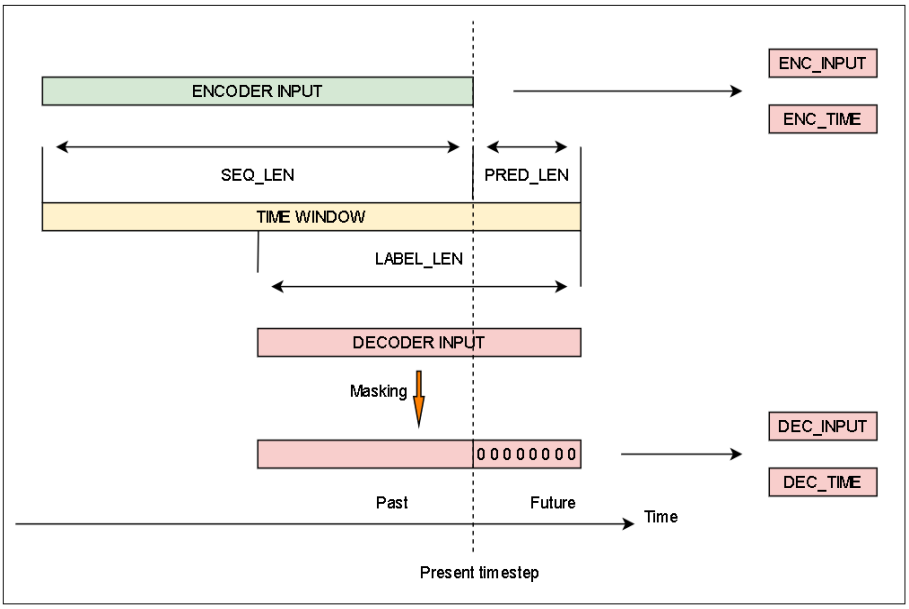


Figure 5.4: Time window split into the four components of the Informer input.

- The *encoder and decoder time inputs*, enclosing the temporal information of the series. These two tensors are built with the same procedure followed for the previously mentioned value ones, but in this case the feature space is substituted with a *time encoding space* of tunable granularity. For a 15 min-scale encoding, which is the one used in the *Informer* model, five *time features* are created, corresponding to *month*, *day*, *weekday*, *hour* and *minute* representations. In this way, each $[1, w, F]$ tensor is mapped into one of shape $[1, w, 5]$. A visualization of time encoding is provided in Fig.5.5.

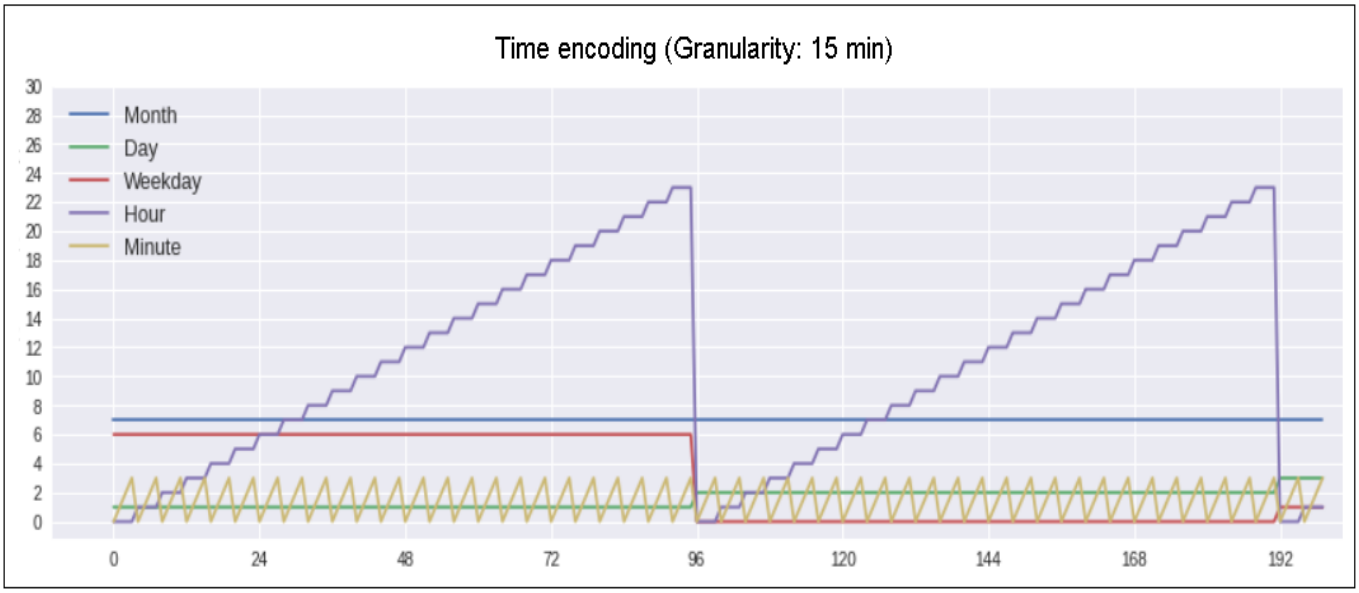


Figure 5.5: Visualization of time encoding corresponding to the time steps between $01/07$ and $02/07$, at a 15 min granularity.

Once created, these four components are further processed by the encoder and decoder embedding layers, described in the following paragraph.

5.3.2 Input embedding layers

The embedding process is equal for both the encoder and the decoder sides. It is carried out by an apposite block, depicted in Fig.5.6, which maps the data into tensors of shape $[1, L_{in}, d_{model}]$, where L_{in} is set to L_s for the encoder and to L_l for the decoder, while d_{model} is the dimension of the internal data representation inside the attention layers.

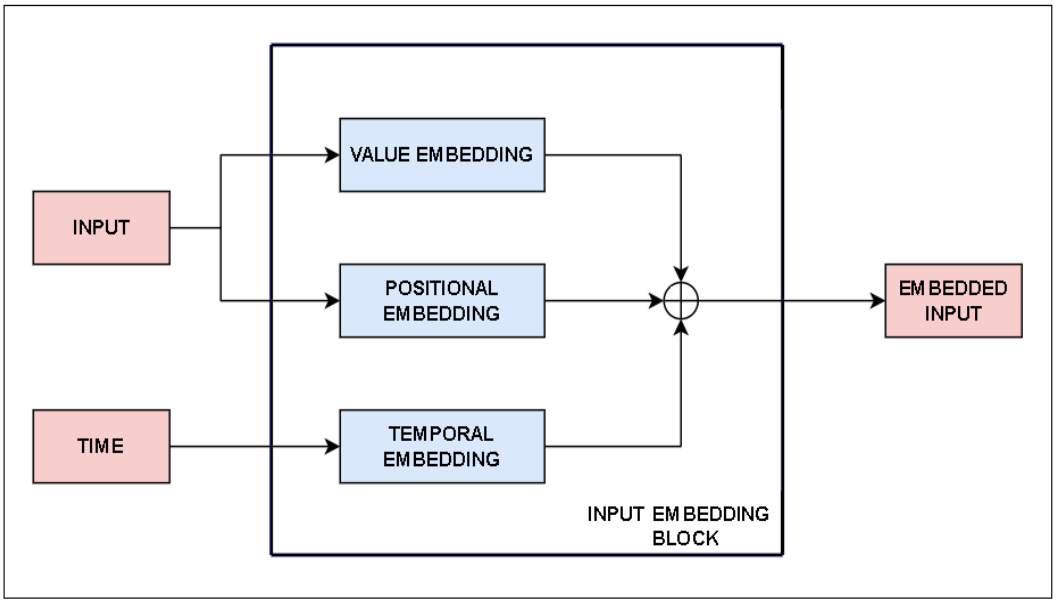


Figure 5.6: Informer architecture.

Taking as input both the value and time tensor elements described in the previous paragraph, each embedding layer outputs the sum of three different components:

- a *value* embedding X_{value}, computed on the value input and represented by a scalar projection of the form:

$$X_{value} = Activation(Conv1D(X)) \tag{5.1}$$

  where the *Leaky ReLU* is adopted as the activation function;

- a *positional* embedding X_{pos}, also applied to the value input and represented by the classical sine/cosine encoding of the vanilla Transformer;

- a *temporal* embedding X_{time}, acting over the time input and represented by the sum of five different linear embeddings of dimension d_{model}, one for each time feature:

$$X_{time} = \sum_{k \in A} LinearEmbedding(x_k) \tag{5.2}$$

with $A = \{month, day, weekday, hour, minute\}$

The final input embedding is then given by:

$$X_{embed} = X_{value} + X_{pos} + X_{time} \tag{5.3}$$

and is a $[1, L_{in}, d_{model}]$ tensor ready to be processed by the subsequent attention layers.

5.3.3 Encoder layers and ProbSparse Attention

The encoder layers of the *Informer*, depicted in Fig.5.7, are structurally similar to the vanilla Transformer ones, being composed by an attention block followed by a feed-forward projection, with residual connections after each of them.

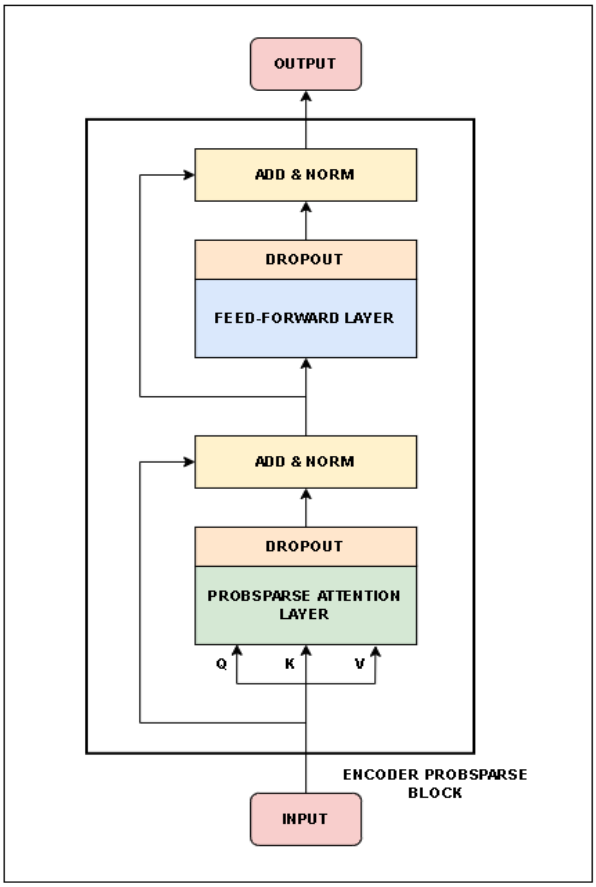


Figure 5.7: Structure of the Informer encoder blocks. With respect to the original Transformer model, the standard attention mechanism is substituted with the *ProbSparse* one.

The main difference with respect to the canonical model resides in the use of *ProbSparse* attention layers, which are able to reduce the time and memory complexity of the attention computation from $O(L_k \cdot L_q)$ to $O(L_q \cdot lnL_k)$ (where L_q and L_k are the number of queries and keys) without a loss in the overall performances.

The idea behind this proposed mechanism is that computing each query-key dot product pair is redundant, since the majority of meaningful information is carried out by only a few elements [47]. For this reason, *ProbSparse* allows each key to only attend to the top-u dominant queries, with $u = c \cdot ln(L_q)$ (where c is an hyperparameter), ranked by means of a *sparsity score function*

M. If Q and K represent the query and key matrices, and $q_i \in Q, k_j \in K$,, the score of each query q_i is given by:

$$M(q_i, K) = \max_j \left(\frac{q_i k_j^T}{\sqrt{d_{model}}} \right) - \frac{1}{L_k} \sum_{j=1}^{L_k} \left(\frac{q_i k_j^T}{\sqrt{d_{model}}} \right) \tag{5.4}$$

in other words, for each query the maximum and the mean value of its scaled dot product with all keys is computed, and the difference of these two components is considered. This peculiar ranking metric is an approximation of how much the probability distribution of the query attention score with respect to the keys is dissimilar to the uniform distribution: the underlying hypothesis is that queries which are dominant in the attention computation show a peak in their distribution (reflecting an ”activation” when coupled to certain keys), while uninteresting ones are associated to a ”flat” plot (producing the same response regardless of their pairing). A detailed formalization of this concept, along with an explanation on how the score function $M(q_i, K)$ is constructed, is provided in Appendix A.

Back to the *ProbSparse* attention computation, we can see that until now the complexity is still $O(L_q \cdot L_k)$, since for each query q_i its dot product $q_i k_j^T$ with all the keys k_j must be computed. It is here that a second simplification is made: instead of considering the full key matrix K, the authors propose to randomly sample $U = c \cdot ln(L_k)$ keys in order to obtain a sparse matrix $\bar{K}$ on which the rows corresponding to non-sampled keys are padded with zeros and thus do not contribute to the score computation. The approximated score function $\bar{M}$, which is the one used in the Informer, becomes:

$$\bar{M}(q_i, \bar{K}) = \max_{k_j \in \bar{K}} \left(\frac{q_i k_j^T}{\sqrt{d_{model}}} \right) - \frac{1}{U} \sum_{k_j \in \bar{K}} \left(\frac{q_i k_j^T}{\sqrt{d_{model}}} \right) \tag{5.5}$$

With this method, only $L_q \cdot lnL_k$ dot-product pairs are computed, thus resulting in a major efficiency gain with respect to the standard attention mechanism.

5.3.4 Conv1D & Pooling layers

After each encoder attention block, except for the last one, a *Conv1D & Pooling* layer is in charge of *distilling* the attention output. This component, whose structure is depicted in Fig.5.8, performs a 1-D convolution (with kernel size = 3) along the time dimension, followed by a layer normalization and an ELU activation function. At the end, a Max Pooling operation, with stride = 2, is applied: this reduces by half the size of data along the feature space. This ”distilling” operation, which is responsible for the funnel-shape structure of the encoder, sharply reduces the overall space complexity and helps discarding redundant information in traversing data.

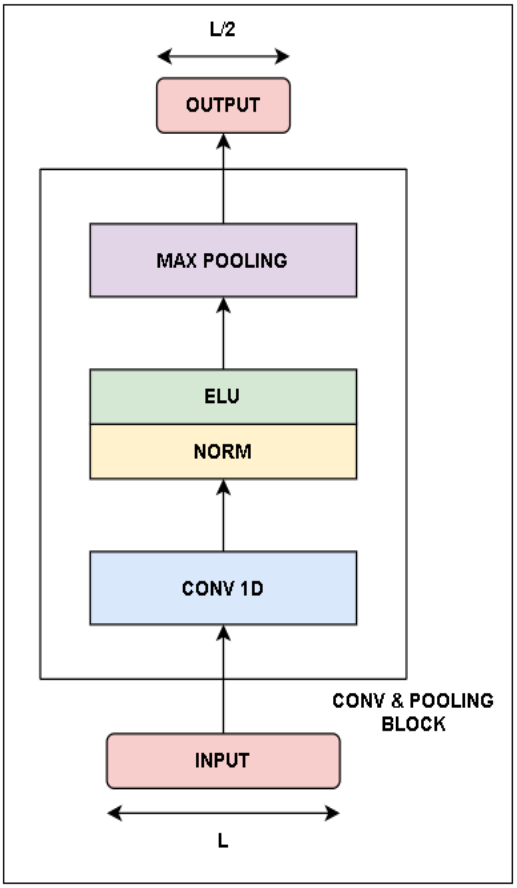


Figure 5.8: Internal architecture of the Conv1D & Pooling layers of the Informer.

5.3.5 Decoder layers and final dense output

Just like the encoder, the decoder layers of the Informer are similar to the original Transformer ones, except for the use of the *ProbSparse attention*. Their structure is depicted in Fig.5.9.

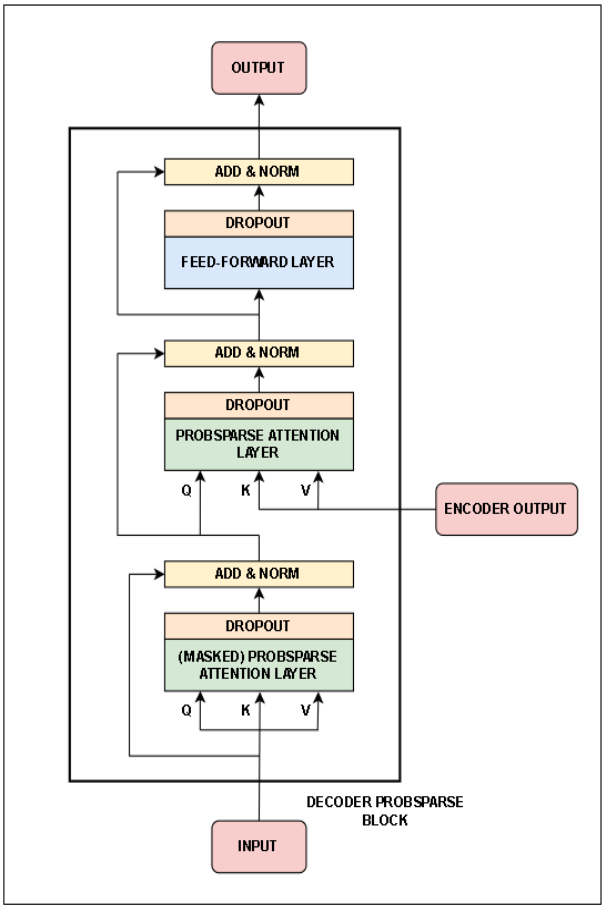


Figure 5.9: Internal structure of the Informer's decoder layers.

Each decoder layer is composed by three parts. The first sub-layer, connected to the embedded decoder input, performs a standard self ProbSparse attention; only in the first decoder layer, this attention is *masked*, preventing the elaboration of future time steps data by the model. The second component is another *ProbSparse* block, computing a cross-attention between the decoder queries and the keys and values provided by the encoder output. At last, a final feed-forward layer is used to project the data outside the block. As usual,

each of these layers is provided with dropout and residual connections.

In the standard implementation, two of these decoder layers are stacked together; at the end of the second one, a final Dense layer is in charge of elaborating the decoder output to produce the final Informer output, represented by the target window of predictions. This output is generated by one forward procedure, rather than the time consuming "dynamic decoding" used in the conventional encoder-decoder architectures, on which the forecasting is done in multiple steps and at each step the previous model output is fed as input for the next prediction. This "one-shot generative inference" allows to significantly reduce the time burden in long-window forecasting applications.

5.3.6 Informer model hyperparameters

The full list of the *Informer* hyperparameters is provided in Tab. 5.3.

Hyperparameter	Description
seq_len	Input sequence length of the encoder
label_len	Portion of lookback window used as input for the decoder
pred_len	Prediction sequence length
Factor	c factor used in the ProbSparse attention
d_{model}	First encoder layer and all decoder layers dimension
N_heads	Number of heads in the Attention layers
enc_layers	Number of encoder layers
dec_layers	Number of decoder layers
d_{ff}	N° of units of the Feed-Forward layers
Dropout	Dropout rate of the model layers

Table 5.3: Hyperparameters table of the Informer model.

Chapter 6

Experiments description and Setup

This chapter will provide a description of the investigations carried out in this thesis work, along with their associated setup and preliminary steps. The experiments can be split in two main categories:

- **Analysis and comparison of the models performances.** The four considered models, namely *CNN*, *LSTM*, *TransformerT2V* and *Informer*, are trained and evaluated on the *ETTm1* and *CU-BEMS* datasets, associated to two different TSF problems. The focus is on studying the effectiveness of transformer-based models, and their comparison with non-transformer ones.
- **Study of the approximations carried out by the ProbSparse attention.** The ProbSparse mechanism of the Informer is able to reduce the complexity of the attention from $O(L_q \cdot L_k)$ to $O(L_q \cdot ln(L_k))$, by involving in the computation only a subset of queries and keys. The experiments of this subgroup are aimed at evaluating the goodness of this approximation, and its relation with the hyperparameter responsible for the number of sampled keys and top queries considered.

The results of these experiments will then be provided in Chapter 7.

6.1 Models training and evaluation on the proposed datasets

This section will describe the preprocessing operations applied to the data, the hyperparameters choice for the models and the training setup followed for the experiments.

6.1.1 Data preprocessing and split

Before training the models, the following preprocessing steps, common to both datasets, have been followed:

- **Filling of missing values.** Timesteps on which one or more feature value is missing have been dealt with a *mean interpolation*, namely the missing element has been approximated with the mean value of its neighbours.

- **Data normalization.** Since the data features are heterogeneous, and come with different scales and units of measure, it is important to map them on a same range of values to assign them equal weight. For this purpose, a *min-max normalization* has been applied:

$$x' = \frac{x - \min x}{\max x - \min x} \tag{6.1}$$

  with this operation, all features are mapped into the $[0, 1]$ interval.

- **Train/validation/test split.** The data have been split into *train*, *validation* and *test* sets, by following a 80%/10%/10% ratio depicted in Fig.6.1.

Figure 6.1: Visualization of ETTm1 and CU-BEMS datasets split into train, validation and test data.

- **Input and label creation.** Once set the lookback window and the forecasting target, the train, validation and test series elements have been sorted to form the models input and the associated ground truth labels (corresponding to the exact prediction values).

6.1.2 Hyperparameters setting

In order to obtain the best results, various hyperparameter choices have been tested for each model, resulting in the final configurations described in Tab. 6.1, 6.2 and 6.3.

As for the forecasting target, for the Informer model a window of 24 steps into the future has been considered, while for the other models two different targets at 12 and 24 steps into the future have been chosen, in order to compare their predictions with the Informer ones.

Model	Hyperparameter	Value
LSTM, CNN	seq_len	128
LSTM, CNN	foresight	12, 24
LSTM	units_lstm	128
LSTM	units_dense_lstm	64
CNN	filters_conv	128
CNN	units_dense_conv	64
CNN	conv_width	5

Table 6.1: Final hyperparameter configuration chosen for the LSTM and CNN models.

TransformerT2V	
Hyperparameter	Value
seq_len	128
foresight	12, 24
d_{model}	256
N_heads	12
FF_dim	256
N_dense	64
Dropout	0.1

Table 6.2: Final hyperparameter configuration chosen for the TransformerT2V model.

Informer	
Hyperparameter	Value
seq_len	96
label_len	48
pred_len	24
Factor	5
d_{model}	512
N_heads	8
enc_layers	3
dec_layers	2
d_{ff}	512
Dropout	0.1

Table 6.3: Final hyperparameter configuration chosen for the Informer model.

6.1.3 Training configuration and schedule

For all the models the training has been carried out with a *batch size* of 32 and a maximum number of *epochs* of 10. The *Adam* optimizer has been used, with a starting *learning rate* of 10^{-4}.

The *loss function* chosen is the *mean squared error* (MSE):

$$MSE(y_{true}, y_{pred}) = \frac{1}{N}\sum_{i=1}^{N}(y_{true} - y_{pred})^2 \tag{6.2}$$

while the evaluation metric is the *mean average error* (MAE):

$$MAE(y_{true}, y_{pred}) = \frac{1}{N}\sum_{i=1}^{N}|y_{true} - y_{pred}| \tag{6.3}$$

As for the training runtime, a custom schedule has been adopted, with two callbacks:

- An *early stopping* callback is in charge of interrupting the model training if, after a certain number of epochs, the validation loss doesn't decrease;
- A *learning rate reduction on plateau* callback decreases the learning rate by a percentage whenever the validation loss stops improving during the training.

The full training configuration is provided in Tab.6.4.

Training configuration	
Batch size	32
Epochs	10
Optimizer	Adam
Starting learning rate	10^{-4}
Early stopping patience	4 epochs
Learning rate plateau reduction patience	2 epochs
Learning rate reduction factor	0.1
Minimum learning rate	10^{-10}

Table 6.4: Training configuration for the proposed models.

6.2 Analysis of the ProbSparse attention

This section will describe the proposed experiments related to a more in-depth analysis of the ProbSparse attention mechanism, and the role of its associated hyperparameter in the resulting approximation. More precisely, it will be reported the reference models on which the analysis takes place, and subsequently the two typologies of investigations carried out: a study of the *approximation in the query score matrix*, and one about the error in the resulting *top-u subset of queries*.

6.2.1 Reference models and aims of the experiments

The models on which the following experiments have been carried out are two *Informer architectures*, trained on the CU-BEMS dataset. The first is a canonical, "sampled" model: its $probsparsefactor$ hyperparameter, or c in a compact notation, determines both the number of top-u queries ($u = c \cdot ln(L_q)$) and the number of sampled keys ($S = c \cdot ln(L_k)$) used to approximate the keys set K with a subset $\bar{K}$. The second model is instead a "full" one: while the number of top-u queries is still determined by c, all the keys are considered and no sampling is made. Using these trained models as a tool, two questions have been asked:

- **How good is the sampling-based approximation of the query-key dot product probability distribution?** Taking into account the "sampled" model, the objective is to determine the difference between the approximated query scores $\bar{M}$ and the equivalent scores M computed by using all the keys. It is also noteworthy to study the consequent difference between the "exact" and the "approximated" ranking orders, and the resulting top-u queries;
- **How well the probsparse mechanism behaves if the distribution is approximated only after the model training?** Starting from the "full" model, we study how the aforementioned query scores and rankings change if the sampling mechanism is applied only after the Informer attention weights are already optimized for a full-keys probsparse attention.

In order to try answering these questions, to both trained Informers is given the same CU-BEMS input, corresponding to a timestep window centered into a peak of electrical consumption: this choice is finalized to induce a strong response into the model layers. Then, for each model, from the first head of the last encoder layer the generated query Q and key K tensors have been

extracted, just before the probsparse mechanism is carried out; obtaining Q and K represents the starting point of the subsequent experiments described in the next paragraphs.

6.2.2 Study of the approximation in the query score matrix

Given a queries set Q and a keys set K, the "full" score of each query $q_i \in Q$ is given by:

$$M(q_i, K) = \max_{k_j \in K} \left(\frac{q_i k_j^T}{\sqrt{d_{model}}} \right) - \frac{1}{U} \sum_{k_j \in K} \left(\frac{q_i k_j^T}{\sqrt{d_{model}}} \right) \tag{6.4}$$

while the approximated, "sampled" score is provided by:

$$\bar{M}(q_i, \bar{K}) = \max_{k_j \in \bar{K}} \left(\frac{q_i k_j^T}{\sqrt{d_{model}}} \right) - \frac{1}{U} \sum_{k_j \in \bar{K}} \left(\frac{q_i k_j^T}{\sqrt{d_{model}}} \right) \tag{6.5}$$

where $\bar{K}$ is a subset of $S = \lceil c \cdot ln(L_K) \rceil$ keys randomly sampled from K. Since for each model the probsparse layer of reference works with 23 queries and keys, c is in the range of integers $[1, 7]$ (with $S = 4$ for $c = 1$ and $S = 22$ for $c = 7$). For each possible value of c, the distance between the real $M(q_i, K)$ and the approximated $\bar{M}(q_i, \bar{K})$ scores has been computed, by using the *root mean square error* as metric:

$$RMSE(M, \bar{M}) = \sqrt{\sum_{i=1}^{L_q} \frac{\left(M(q_i, K) - \bar{M}(q_i, \bar{K}) \right)^2}{L_q}} \tag{6.6}$$

Since the subset $\bar{K}$ is random (due to the random keys sampling), this procedure has been repeated N times, with N sufficiently large (in these experiments, $N = 1000$), and the *mean RMSE value* has been taken as the final result.

Furthermore, the two main components of the score function have been evaluated separately. Recalling Eq.6.4, $M(q_i, K)$ can be seen as:

$$M(q_i, K) = MAX(q_i, K) - MEAN(q_i, K) \tag{6.7}$$

with

$$MAX(q_i, K) = \max_{k_j \in K} \left(\frac{q_i k_j^T}{\sqrt{d_{model}}} \right) \tag{6.8}$$

and

$$MEAN(q_i, K) = \frac{1}{L_K} \sum_{k_j \in K} \left(\frac{q_i k_j^T}{\sqrt{d_{model}}} \right) \tag{6.9}$$

where $MAX(q_i, K)$ represents the peak of the query-keys dot product distribution, while $MEAN(q_i, K)$ its average value. Given their approximated counterparts $\bar{MAX}(q_i, \bar{K})$ and $\bar{MEAN}(q_i, \bar{K})$, the corresponding mean RMSE values have been computed for each possible value of c.

6.2.3 Study of the approximation in the query ranking

Since the probsparse attention output is not directly influenced by the $M(q_i, K)$ scores, but only by the top-u queries choice, it has also been decided to directly measure the distance between the two query rankings $R = [q_1^R, ..., q_{L_q}^R]$ and $\bar{R} = [q_1^{\bar{R}}, ..., q_{L_q}^{\bar{R}}]$ obtained from M and $\bar{M}$. The rationale behind this is that two different sets of scores could determine two equal query orderings, and consequently the same final result: therefore, regardless of the error between M and $\bar{M}$, if R and $\bar{R}$ are similar enough the approximation obtained by considering only the subset of keys $\bar{K} \in K$ can be deemed valid.

The relation between R and $\bar{R}$ has been observed by means of two different points of view, each measured with a corresponding metric:

- **Queries ordering.** This case aims to measure how many queries are placed at the same position in both rankings, considering both the *full*

ranking and the *top-u only*. The proposed metric is the *normalized Hamming distance*, computed as follows:

$$H(R, \bar{R}) = \frac{\sum_{i=1}^{N} f(R[i], \bar{R}[i])}{N} \tag{6.10}$$

with

$$f(R[i], \bar{R}[i]) = \begin{cases} 1, & R[i] = \bar{R}[i] \\ 0, & R[i] \neq \bar{R}[i] \end{cases} \tag{6.11}$$

This metric can be applied also for the *top-u only* evaluation, since it does not require the two top-u subsets to share the same queries (although ordered differently).

- **Queries presence in top-u.** Since the probsparse attention output is influenced only by the choice of which queries are involved in the computation and not by their relative ranking, it has been decided to consider the top-u positions of R and $\bar{R}$ as two unordered sets, and measure their similarity by means of their intersection and union only. The proposed metric is the *Jaccard distance*, defined as:

$$D(R_{top-u}, \bar{R}_{top-u}) = 1 - J(R_{top-u}, \bar{R}_{top-u}) \tag{6.12}$$

where $J(R_{top-u}, \bar{R}_{top-u})$ is the *Jaccard similarity index*:

$$J(R_{top-u}, \bar{R}_{top-u}) = \frac{R_{top-u} \cap \bar{R}_{top-u}}{R_{top-u} \cup \bar{R}_{top-u}} \tag{6.13}$$

which represents the *intersection-over-union* between the two considered subsets.

As for the previous set of experiments, since the ”approximated” ranking $\bar{R}$ depends on the randomly sampled keys, the proposed metrics have been computed over $N = 1000$ repeated trials, and their mean value has been taken as the final result.

It is important to underline that the Jaccard distance between the two top-u sets is bound to reach zero with $c = 7$: since in the original architecture this hyperparameter is responsible for both sampled keys and top-u queries numbers, for a maximum value of c all the queries are in top-u, and thus the ”full” and ”sampled” corresponding unordered sets are equal. For this reason, it has been decided to also consider the case in which *c is decoupled into two different hyperparameters* c_k and c_q, one for keys and one for queries: in this way, it is possible to determine the approximation error also for high numbers of sampled keys, with respect to few top-u positions considered. Furthermore, it is possible to study if splitting c into two components could be beneficial for the model performances.

Chapter 7

Results

This chapter will provide the results obtained by the CNN, LSTM, TransformerT2V and Informer models on the ETTm1 and CU-BEMS datasets, and the outcome of the studies on the probsparse mechanism of the Informer.

7.1 Models performances on ETTm1 Dataset

The models performances, in terms of MSE and MAE metrics, on the normalized ETTm1 test dataset are depicted in Tab.7.1.

Model	MSE (t+12)	MAE (t+12)	MSE (t+24)	MAE (t+24)
CNN	0.0015	0.0294	0.0022	0.0355
LSTM	0.0016	0.0291	0.0035	0.0442
TransformerT2V	0.0019	0.00327	0.045	0.0511
Informer	0.0007	0.0193	0.0011	0.0218

Table 7.1: Models results for the ETTm1 test data.

From the metrics values it can be observed that all models perform very well on the ETTm1 dataset, with the *Informer* architecture performing best while the *TransformerT2V* having performances comparable with the CNN and LSTM ones. This could suggest that, for low-feature datasets, the vanilla

Attention mechanism plus the introduction of a time encoding does not provide significant advantages over standard methods such as convolutions and recurrence; another hypothesis is that discarding the decoder component of the Transformer could have hindered the advantages provided by the vanilla architecture.

The situation is different for the *Informer model*, outperforming other architectures by a significant margin and obtaining results similar to the ones achieved by the model authors on the same dataset [47].

A visualization of each model's forecasting on the ETTm1 test set is depicted in Fig.7.1. Overall, all the predictions manage to follow the series trend, with some oscillations especially in the *TransformerT2V* case. The *Informer* forecasting is instead very precise and seems to capture very well the local maxima and minima of the series.

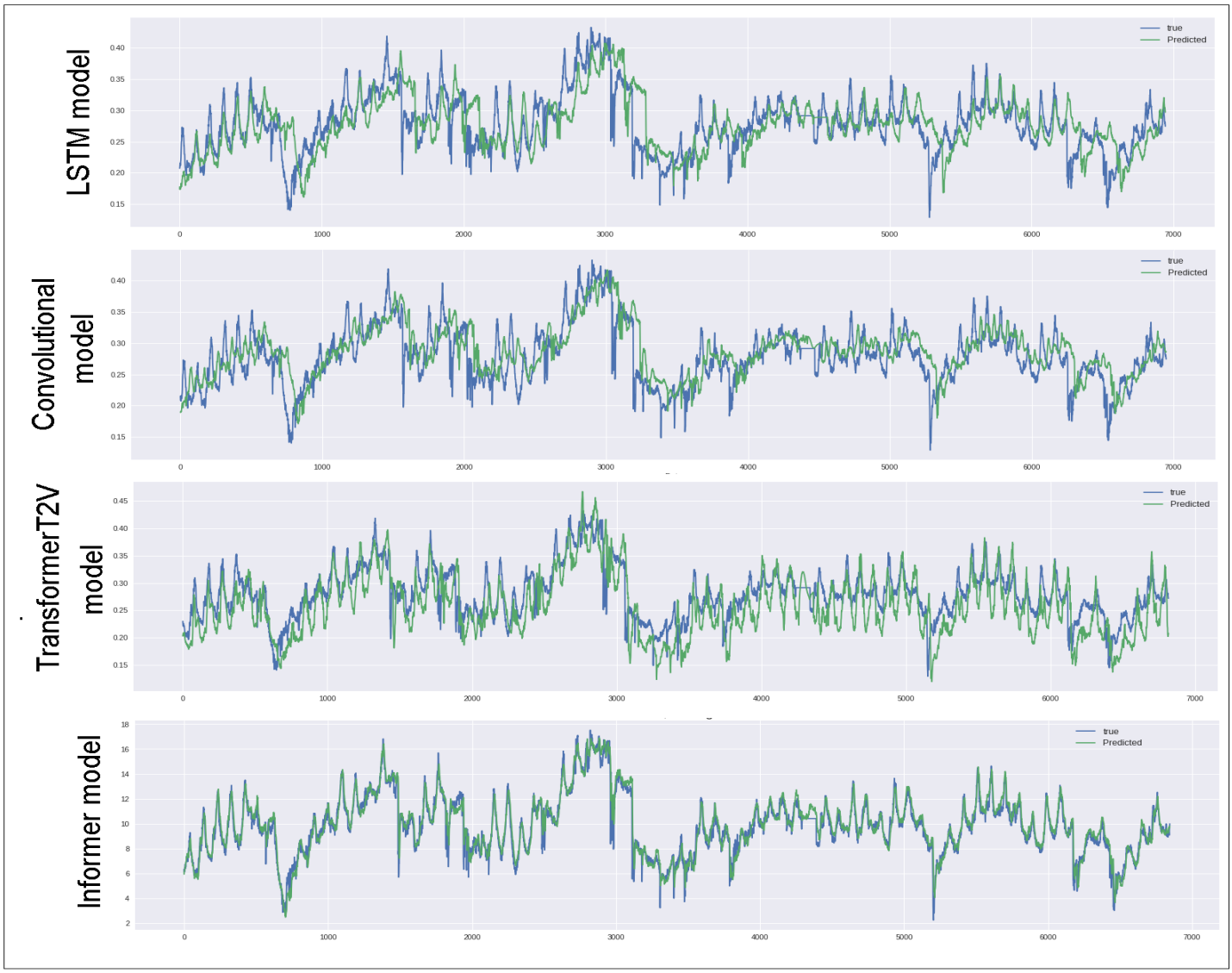


Figure 7.1: ETTm1 test set predictions for the LSTM, CNN, TransformerT2V and Informer architectures.

7.2 Models performances on CU-BEMS Dataset

The models performances, in terms of MSE and MAE metrics, on the normalized CU-BEMS test dataset are depicted in Tab.7.2.

Model	MSE (t+12)	MAE (t+12)	MSE (t+24)	MAE (t+24)
CNN	0.0411	0.1466	0.0437	0.1498
LSTM	0.0317	0.1067	0.0368	0.1103
TransformerT2V	0.0241	0.0791	0.0315	0.1027
Informer	0.0104	0.0391	0.0197	0.0408

Table 7.2: Models results for the CU-BEMS test data.

In this case, the transformer-based models perform significantly better with respect to non-transformer ones. This could be due to the involvement of a much higher number of features, which simpler models struggle to keep track of.

As for the previous dataset, a visualization of each model's predictions on the CU-BEMS test set is depicted in Fig.7.1.

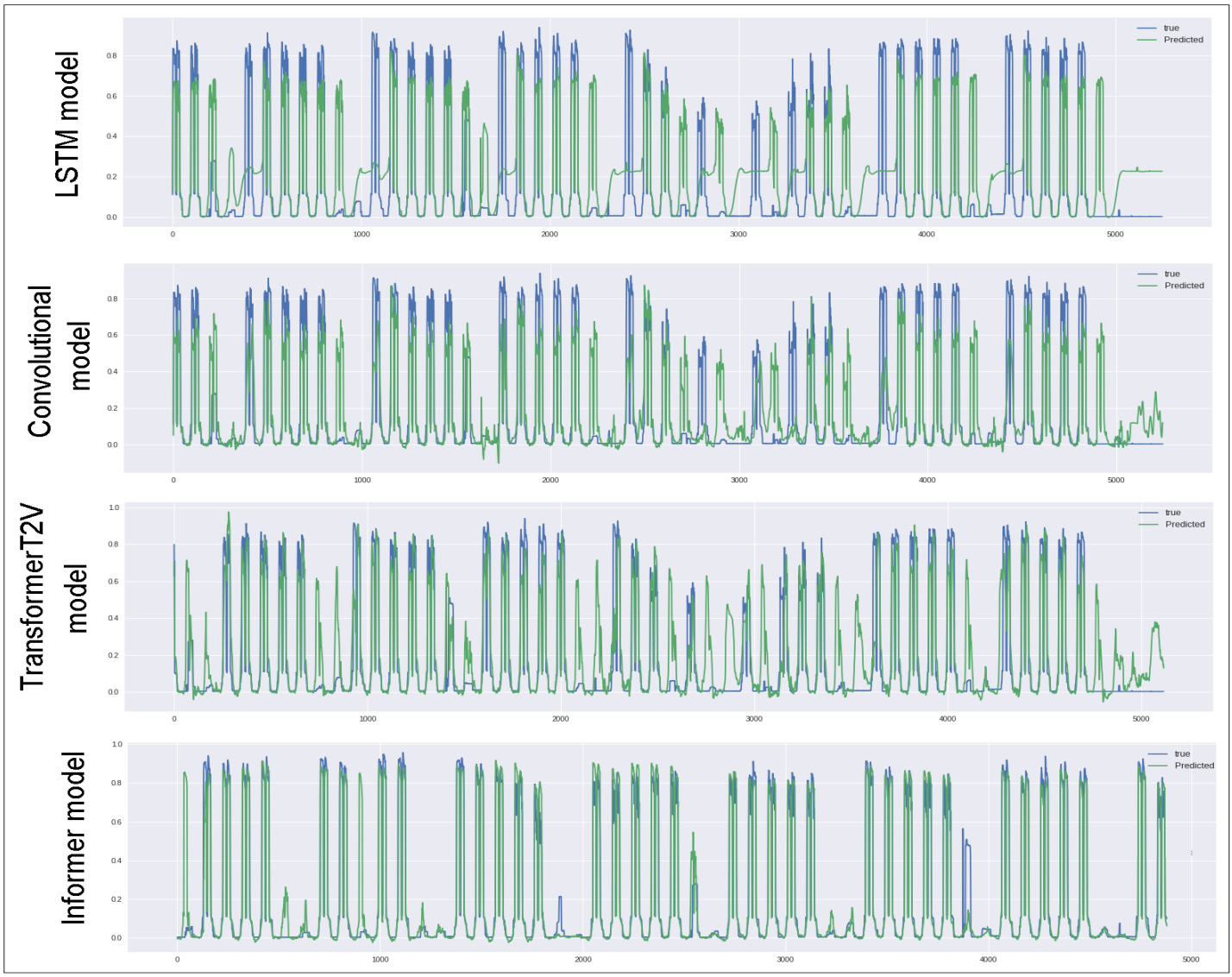


Figure 7.2: CU-BEMS test set prediction for the LSTM, CNN, TransformerT2V and Informer architectures.

The *Informer* is still the best performing architecture, with low MSE and MAE scores. Still, it struggles to correctly predict time steps related to festivities, especially in the case of predictions far in the future. An example of this is depicted in Fig.7.3.

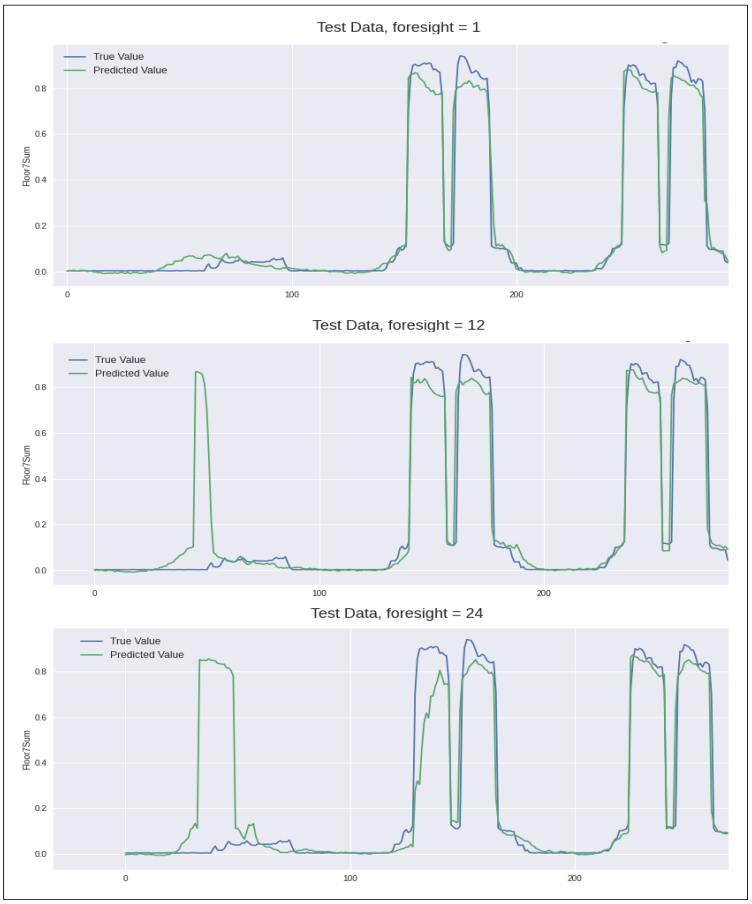


Figure 7.3: Example of ”holiday outlier” and related Informer perdiction at 1, 12 and 24 time steps in the future.

In order to determine if the introduction of the ”weekend/holiday” feature in the CU-BEMS dataset is really beneficial for the Informer’s performances, the same model has been trained again it, and the MSE and MAE values have been computed separatedly for working days and weekend/holidays. The results are provided in Tab.7.3.

	All predictions	Working days	Weekend/Holidays
Weekend/holiday used	$MSE = 0.0197$ $MAE = 0.0408$	$MSE = 0.0129$ $MAE = 0.0287$	$MSE = 0.0861$ $MAE = 0.0612$
Weekend/holiday not used	$MSE = 0.0223$ $MAE = 0.0681$	$MSE = 0.0137$ $MAE = 0.0315$	$MSE = 0.1153$ $MAE = 0.1132$

Table 7.3: MSE and MAE scores for predicted data at timesteps $t+24$, depending on whether the related feature column is used or not. The metrics are also computed on timesteps corresponding to working days and weekends/holidays separately.

From the table, it can be seen that while global and working days metrics stay more or less the same, a small improvement is made on the weekend/holidays error, suggesting the beneficial effects of this feature on the overall training.

7.3 Results on the study of ProbSparse Attention

7.3.1 RMSE between query scores

The mean RMSE values between the exact query scores $M(q_i, K)$ and the approximated ones $\bar{M}(q_i, \bar{K})$, computed over 1000 iterations and as a function of the probsparse factor c, are depicted in Fig.7.4 for the ”Full” model, and in Fig.7.5 for the ”Sampled” one; the same figures also provide the results of the investigation focused on the ”max” and ”mean” components of the ranking function.

Factor	M	MAX_M	MEAN_M
1	9.180916	9.062960	1.047607
2	6.436026	6.374761	0.979152
3	4.536623	4.583366	0.940638
4	3.560030	3.417436	0.955880
5	2.547962	2.458380	0.990459
6	2.189097	2.099034	1.062555

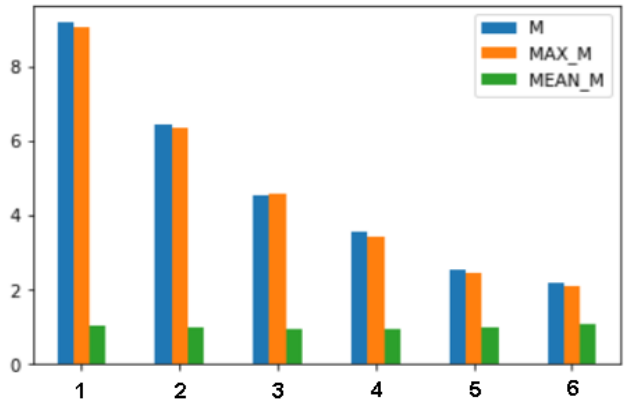
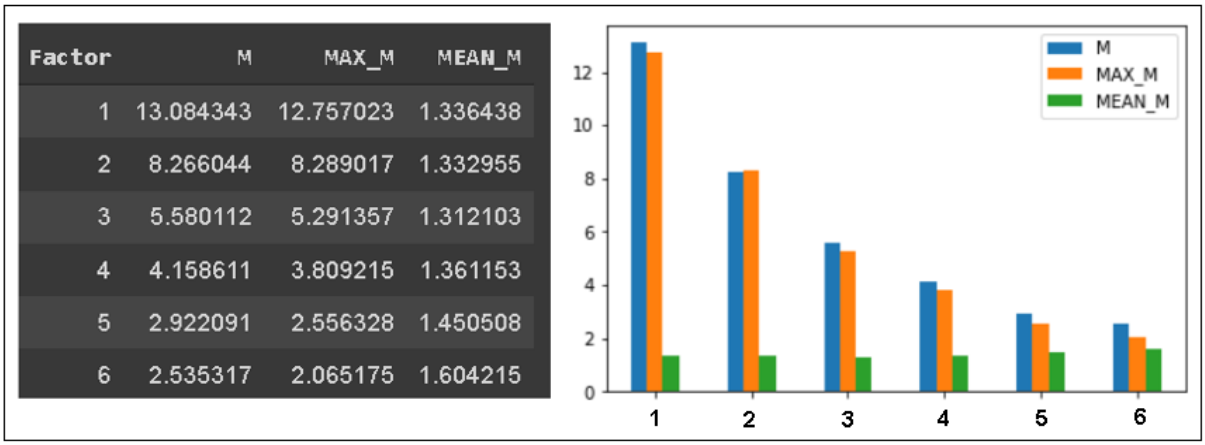


Figure 7.4: RMSE values related to the ranking function investigation on the ”Full” model.



Factor	M	MAX_M	MEAN_M
1	13.084343	12.757023	1.336438
2	8.266044	8.289017	1.332955
3	5.580112	5.291357	1.312103
4	4.158611	3.809215	1.361153
5	2.922091	2.556328	1.450508
6	2.535317	2.065175	1.604215

Figure 7.5: RMSE values related to the ranking function investigation on the ”Sampled” model.

From the RMSE tables and their associated bar charts some considerations could be made. First of all, while the error starts higher for low values of c in the ”sampled” model, in both cases tends to reach the same plateau for high c values, with a similar descending curve; as expected, the two differently trained models show the same behaviour for high values of the hyperparameter, but even for lower values of the latter their difference is not so marked.

By looking at the *MAX* and *MEAN* components, it is possible to see that in both cases the RMSE of the latter is relatively low, and almost constant regardless of c, while the first starts high and decreases progressively: this suggests that even by sampling a few number of keys the mean value of the distribution is approximated well, while its maximum is not. The fact that c only influences the *MAX* component approximation could lead to the suggestion of modifying the original $M(q_i, K)$ score function in order to give it a major weight in the final result, for example by discarding the *mean* component from the computation.

Still, only looking at the error in the score values is not enough to draw strong conclusions, since different query scores not necessarily lead to different rankings.

7.3.2 Hamming distance between query rankings

The following tables (Tab.7.4, Tab.7.5) show the normalized Hamming distances between query rankings built from the exact M and approximated $\bar{M}$ scores, considering both the full L_q queries and the top-u only. Their associated bar charts are also depicted in Fig.7.6

	”Full” Model	
Factor	Hamming distance, full ranking	Hamming distance, top-u ranking
1	0.92	0.84
2	0.87	0.90
3	0.83	0.71
4	0.78	0.69
5	0.74	0.56
6	0.26	0.72

Table 7.4: Normalized Hamming distance between query rankings in the ”Full” model. ”Full ranking” refers to the full query ordering, while ”top-u ranking” the one of top-u queries only

	”Sampled” Model	
Factor	Hamming distance, full ranking	Hamming distance, top-u ranking
1	0.90	0.84
2	0.76	0.64
3	0.62	0.66
4	0.51	0.59
5	0.46	0.37
6	0.44	0.44

Table 7.5: Normalized Hamming distance between query rankings in the ”Sampled” model. ”Full ranking” refers to the full query ordering, while ”top-u ranking” the one of top-u queries only

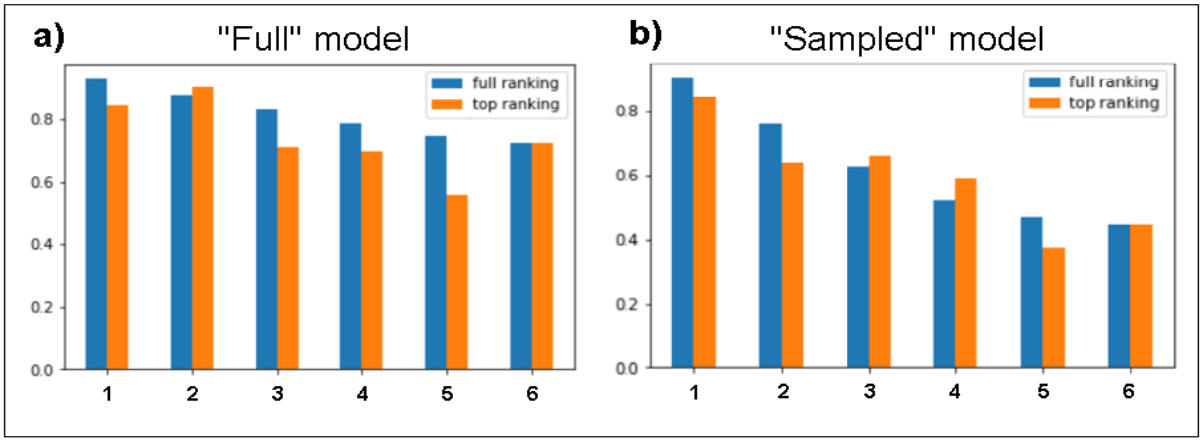


Figure 7.6: Bar charts of the Hamming distance value as a function of c for the ”Full” (a) and the ”Sampled” (b) models.

Unlike in the previous experiment, here the ”full” and the ”sampled” models present different behaviours: the first shows an overall high error in using sampled keys to approximate the queries ranking even for high values of c, while the second performs much better in this sense. In fact, for the ”sampled” model, the ranking approximation error decreases almost linearly with increasing values of c, while for the ”full” one it does not decrease significantly; this suggests that pruning the key distribution information only after the training is not as effective as employing that strategy during it.

Still, for both models the approximation error is relatively high, with even the ”prob” model’s best configuration staying over a 0.35 distance score. This does not necessarily lead to errors in the final attention output, since the probsparse mechanism treats the chosen queries as an unordered set; the following experiment is focused on this aspect.

7.3.3 Jaccard distance between top-u query sets

The Jaccard distances between real and approximated top-u queries sets for the two studied models are depicted in Fig.7.7.

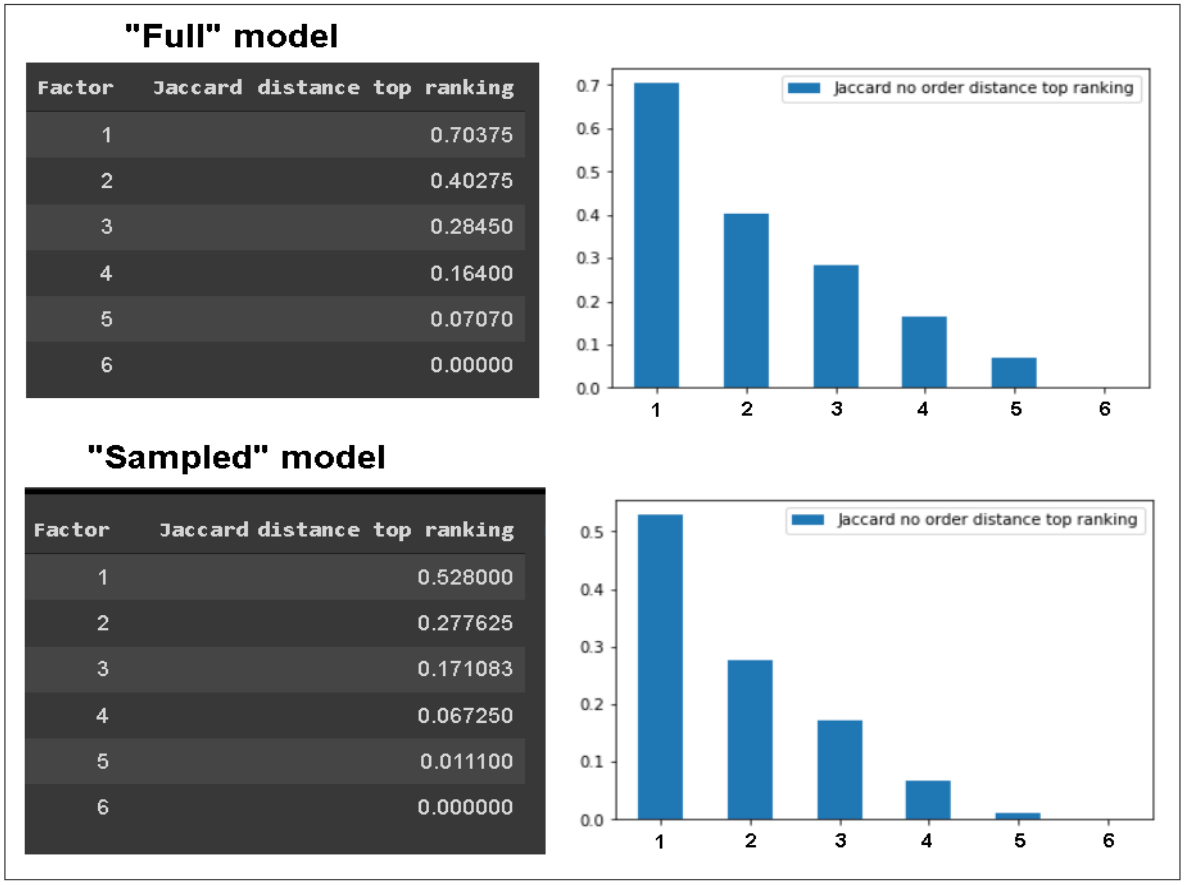
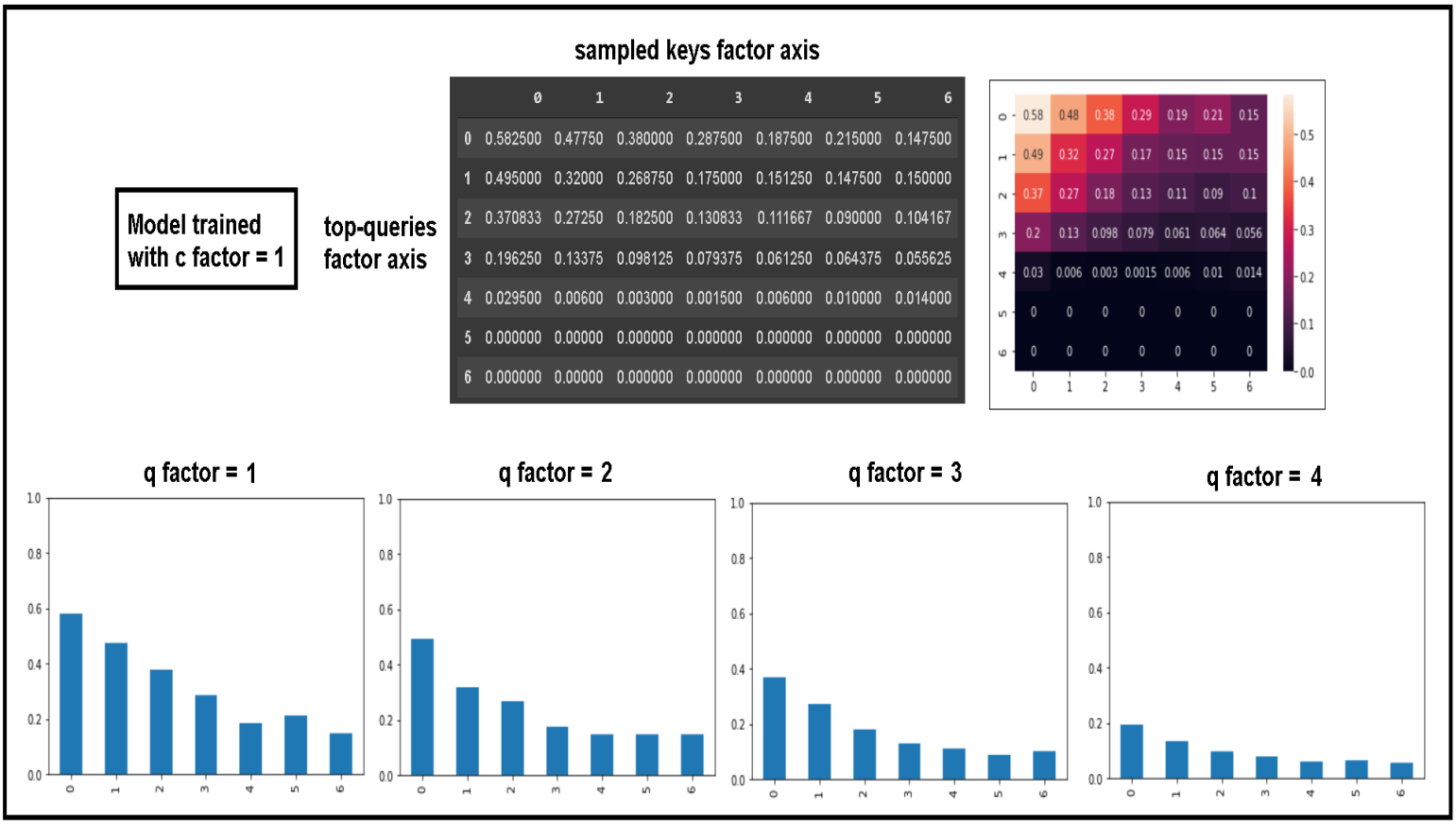


Figure 7.7: Jaccard distance between exact and approximated top-u queries sets for both ”full” and ”sampled” Informer models.

It can be seen that for appropriate values of c the error drops considerably; recalling Table 7.4, for certain values of c onwards, the choice of queries to involve in the attention computation is similar in both the exact and the approximated computations, even if the corresponding Hamming distance is high. This holds for both models, but is particularly true for the ”sampled” one, since the initial Jaccard distance is around 0.5 for the minimum value of c (and so for a small number of sampled keys). Again, this shows the importance of enacting the sampling mechanism during the model training.

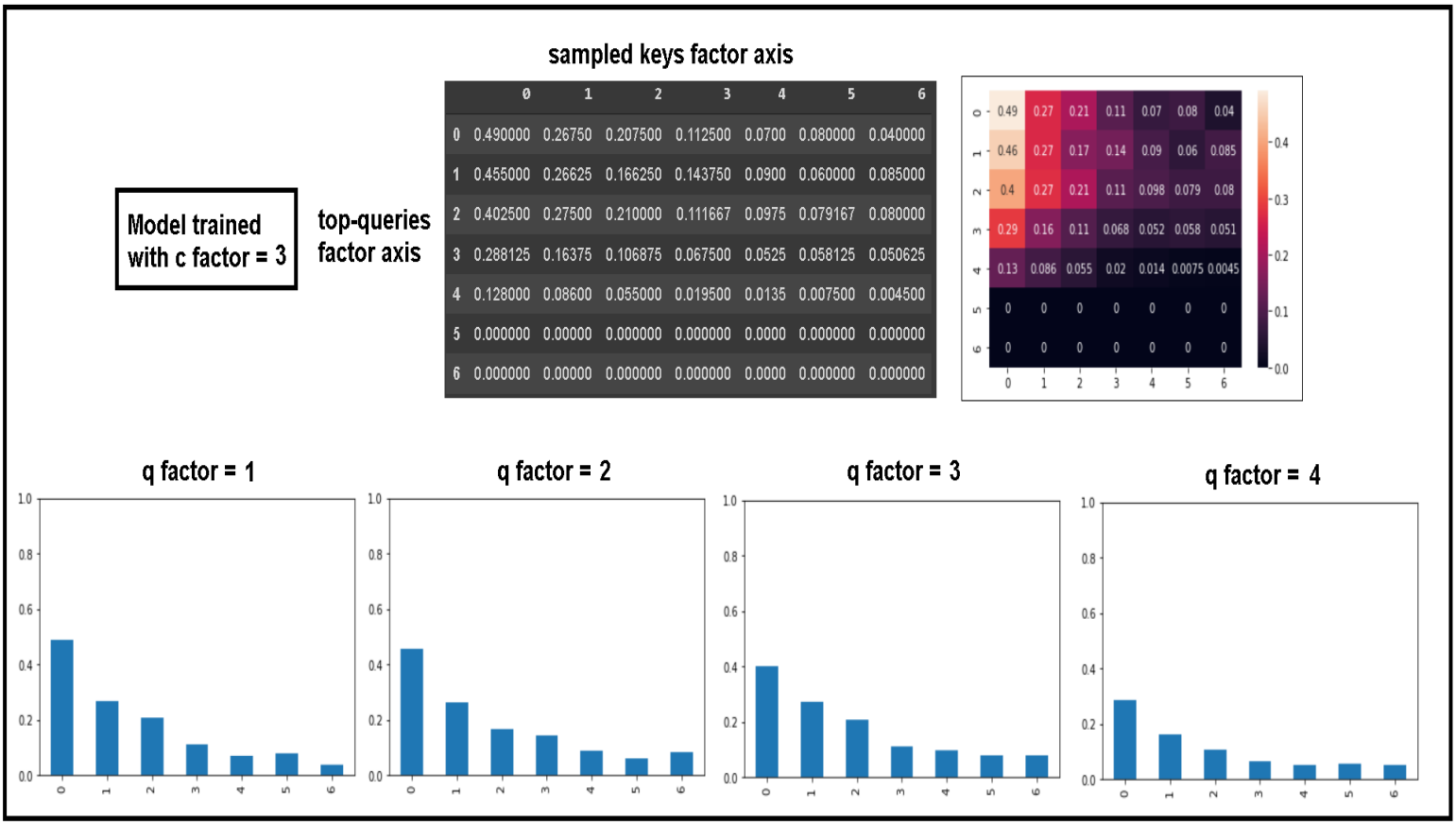
As previously underlined, with this setup the Jaccard distance is bound to reach zero for the the maximum value of c, since in this limit case all queries

are considered in top-positions; this is a consequence of the fact that the prob-sparse factor c is associated to both queries and keys extraction. The effects of decoupling c into two sub-hyperparameters c_q and c_k, of which the first is responsible for the top-u queries and the second for the sampled keys, are described by the last experiment's results, reported below: they show the Jaccard distance matrix between top-u sets, which contains the metric scores associated to all possible c_q and c_k configurations, for three different Informer models trained with $c = 1$ (Fig.7.8), 3 (Fig.7.9) and 5 (Fig.7.10) respectively:



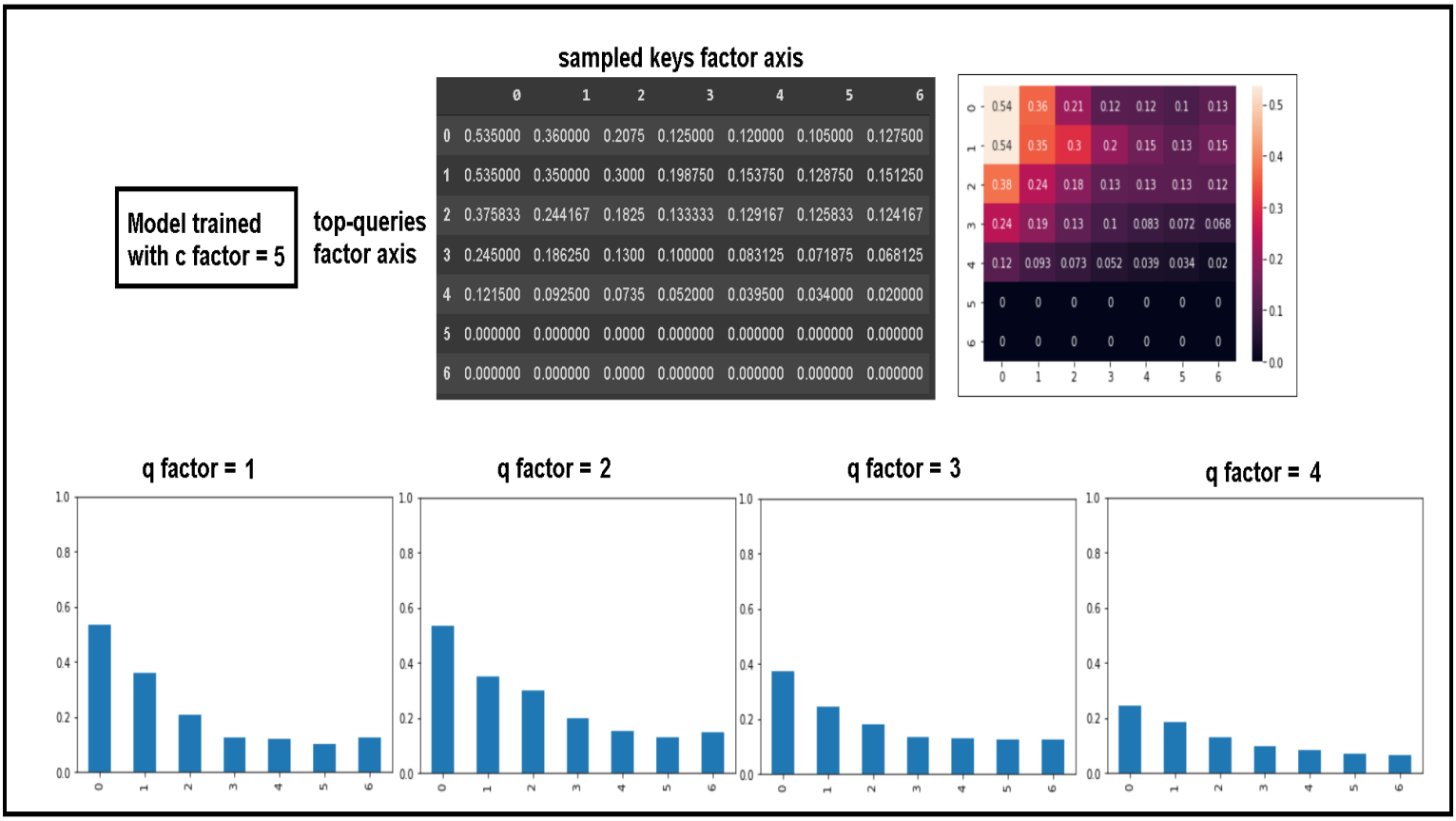
	0	1	2	3	4	5	6
0	0.582500	0.47750	0.380000	0.287500	0.187500	0.215000	0.147500
1	0.495000	0.32000	0.268750	0.175000	0.151250	0.147500	0.150000
2	0.370833	0.27250	0.182500	0.130833	0.111667	0.090000	0.104167
3	0.196250	0.13375	0.098125	0.079375	0.061250	0.064375	0.055625
4	0.029500	0.00600	0.003000	0.001500	0.006000	0.010000	0.014000
5	0.000000	0.00000	0.000000	0.000000	0.000000	0.000000	0.000000
6	0.000000	0.00000	0.000000	0.000000	0.000000	0.000000	0.000000

Figure 7.8: Jaccard distance matrix associated to all possible c_q and c_k configurations, along with the relative heatmap and rows bar charts, for an Informer model trained with $c = c_q = c_k = 1$.



	0	1	2	3	4	5	6
0	0.490000	0.26750	0.207500	0.112500	0.0700	0.080000	0.040000
1	0.455000	0.26625	0.166250	0.143750	0.0900	0.060000	0.085000
2	0.402500	0.27500	0.210000	0.111667	0.0975	0.079167	0.080000
3	0.288125	0.16375	0.106875	0.067500	0.0525	0.058125	0.050625
4	0.128000	0.08600	0.055000	0.019500	0.0135	0.007500	0.004500
5	0.000000	0.00000	0.000000	0.000000	0.0000	0.000000	0.000000
6	0.000000	0.00000	0.000000	0.000000	0.0000	0.000000	0.000000

Figure 7.9: Jaccard distance matrix associated to all possible c_q and c_k configurations, along with the relative heatmap and rows bar charts, for an Informer model trained with $c = c_q = c_k = 3$.



	0	1	2	3	4	5	6
0	0.535000	0.360000	0.2075	0.125000	0.120000	0.105000	0.127500
1	0.535000	0.350000	0.3000	0.198750	0.153750	0.128750	0.151250
2	0.375833	0.244167	0.1825	0.133333	0.129167	0.125833	0.124167
3	0.245000	0.186250	0.1300	0.100000	0.083125	0.071875	0.068125
4	0.121500	0.092500	0.0735	0.052000	0.039500	0.034000	0.020000
5	0.000000	0.000000	0.0000	0.000000	0.000000	0.000000	0.000000
6	0.000000	0.000000	0.0000	0.000000	0.000000	0.000000	0.000000

Figure 7.10: Jaccard distance matrix associated to all possible c_q and c_k configurations, along with the relative heatmap and rows bar charts, for an Informer model trained with $c = c_q = c_k = 5$.

The results show that, regardless of the choice of c for the training, a common pattern can be observed: depending on the query factor c_q, from a certain key sampling parameter c_k onwards a plateau is reached, namely the Jaccard distance does not decrease significantly by increasing c_k. This represents a noteworthy observation, since for certain configurations it is possible to decrease the number of sampled keys, and thus the overall computational burden, with negligible performance degradations.

Chapter 8

Conclusions

8.1 Final remarks

In this thesis work, two groups of investigations have been carried out: the use of *transformer-based architectures for time series forecasting*, and the *analysis of the keys sampling mechanism behind the Informer's probsparse attention*, along with the suggestion of some improvement ideas.

For the first group, two transformer-based architectures, namely the *TransformerT2V* and the *Informer*, have been applied to two different time series forecasting problems, and their performances have been compared with the ones of two classical non-transformer models used in TSF, namely a *CNN* and a *LSTM*. The obtained results show that, while all the proposed models performed very well on both *ETTm1* and *CU-BEMS* datasets, the simple attention-based TransformerT2V performs slightly worse than the non-transformer reference models on ETTm1, and slightly better on CU-BEMS. This suggests that, in the TSF domain, the attention mechanism's benefits show up in the elaboration of high-dimensional data, namely when the dataset contains an high number of features (this is the case of CU-BEMS), while for small dimensions the performances are comparable to the ones obtained by

classical methods. As expected, of all the models the *Informer* is the best performing one, outclassing the prediction accuracy of the other considered architectures by a significant margin: this shows the potential benefits of adopting SOTA transformer-based models for TSF-related applications.

The second group of experiments aimed instead at studying the mechanisms behind the distinctive characteristic of the *Informer*, which is the *Probsparse attention*, and the role of the probsparse hyperparameter c, responsible for both the number of sampled keys and the queries involved in the attention computation. The obtained results showed how variations in the choice of c only affect a component of the query score function, suggesting a rework of the latter in order to be more easily controlled by the hyperparameter. Furthermore, it has been shown how a decoupling of c into two distinct components could prove beneficial, diminishing the computational burden without a loss in the performances. At last, it has been shown how, from certain values of c onwards, the accuracy of the probsparse's internal representations reach a plateau: this could be exploited by fixing a threshold T_h in the approximation error, and tuning the value of c in order to have the smallest number of sampled keys while staying under T_h.

8.2 Future work

Regarding the models performances on the proposed TSF problems, the reasons behind the uneffectiveness of the *TransformerT2V* architectures could be explored further. In particular, its average-to-below-average performances could be due to the lack of a *Decoder*, a key component of many transformer-based models, which has been cut off for the sake of efficiency and to study the sole contributions of the vanilla attention mechanism and the T2V encoding to the overall result. In this line, possible future works could be the introduction of T2V in the original Transformer model, or the substitution on the latter of the canonical attention with different mechanisms, in order to evaluate the

effective importance of an encoder-decoder structure.

As for the Informer model, the experiments on the *probsparse attention* mechanism represent a preliminary analysis, carried out only on the first head of a single encoder layer, and should be supported by more data. Further studies should extend the analysis to all the heads of all layers, in order to evaluate the effectiveness of probsparse's internal approximations on the various model components, and their correlations. For instance, it could be noteworthy analyzing the probsparse mechanism in the decoder's cross-attention layers, on which the input comes from both the encoder and the previous decoder layers.

Other next researches could reside in testing the effects of the proposed modifications, namely the use of a different query score function and the decoupling of the probsparse factor c into two sub-parameters c_q and c_k, on the overall Informer performances. Regarding this, a *runtime-tuning* of the keys sampling factor c_k could be enacted, for instance by means of a reinforcement-learning mechanism which at each time step increases or decreases c_k if the corresponding error metric is over or under a fixed threshold T_h. Furthermore, with this mechanism each attention layer could tune its own parameter value, with possible performance benefits; a future study could determine if this is actually the case.

Appendix A

Foundations of the ProbSparse Attention mechanism

Recalling the canonical Attention computation, represented by the equation

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{A.1}$$

we can see that it represents a *weighted sum* of input values, on which the weights are computed starting from a softmax function applied to scaled dot-products between pairs of queries and keys. From this consideration, the ProbSparse mechanism of the Informer lays its foundations on the hypothesis that the aforementioned softmax scores follow a *long-tail distribution*, depicted in Fig.A.1: only a few dot-product pairs contribute to the major attention computation, while most of the others could be ignored without a significant variation of the final result.

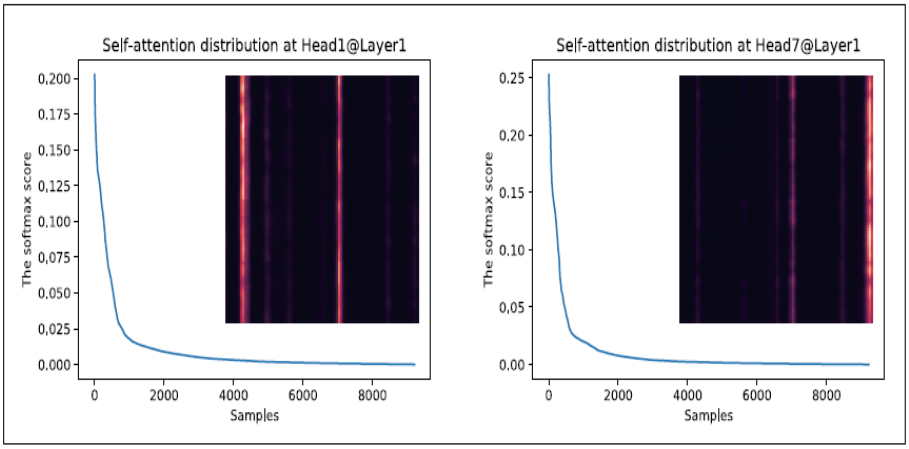


Figure A.1: Long-tail distribution of softmax scores in the canonical Transformer self-attention (Image from [47]).

In this context, the attention equation can be reformulated as follows:

$$Attention(Q, K, V) = \sum_{J} \frac{Ker(q_i, k_J)}{\sum\limits_{l \in L_k} Ker(q_i, k_l)} V_J \tag{A.2}$$

where $Ker(q_i, k_J))$ is an *asymmetric exponential kernel*:

$$Ker(q_i, k_J) = e^{\frac{q_i k_j^T}{\sqrt{d}}} \tag{A.3}$$

In eq.A.2, the elements of the summation can be seen as the probability distribution $p(q_i, k_J)$, for each query q_i, of its attention score with respect to all keys $k_j \in K$ (with $|K| = L_K$):

$$p(q_i, k_J) = \frac{Ker(q_i, k_J)}{\sum\limits_{l \in L_k} Ker(q_i, k_l)} \tag{A.4}$$

For a given query, if its distribution p is similar to the uniform distribution:

$$q(q_i, k_J) = \frac{1}{L_K} \tag{A.5}$$

the query's contribution is trivial and can be discarded. A visualization of this idea is depicted in Fig.A.2.

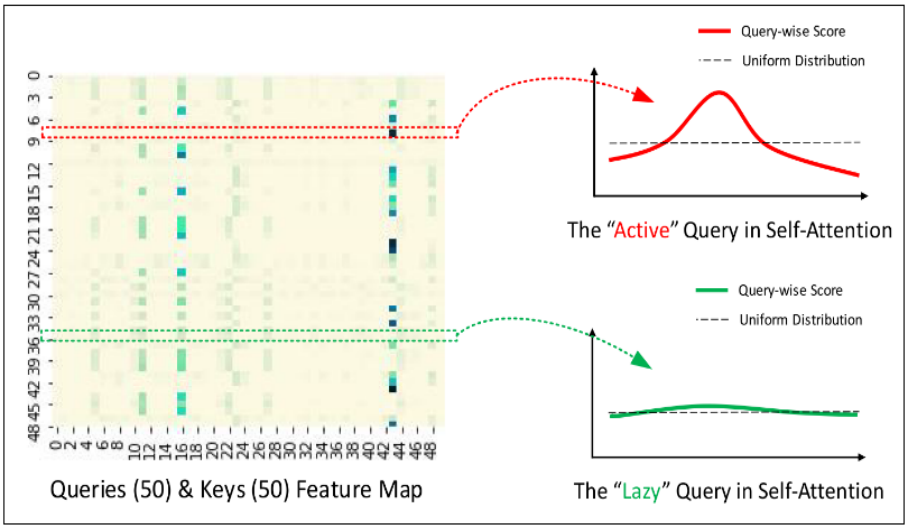


Figure A.2: Probability distribution of dot-product values for an "active" query and a "lazy" one. Active queries show an activation peak in corrispondence to certain keys, while unimportant ones are associated to an uniform response (Image from [47]).

In order to measure the similarity between $p(q_i, K)$ and $q(q_i, K)$, a candidate metric is represented by the *Kullback–Leibler divergence*:

$$KL\left(q||p\right) = ln\left(\sum_{l=1}^{L_K} e^{\frac{q_i k_l^T}{\sqrt{d}}}\right) - \frac{1}{L_K}\sum_{J=1}^{L_K}\frac{q_i k_l^T}{\sqrt{d}} - ln(L_K) \tag{A.6}$$

where, for a given query q_i, the first term is the *log-sum-exp* (LSE) function computed on all keys, the second is the *arithmetic mean*, and the third is a constant that can be discarded from the final result. If the value of $KL\left(q||p\right)$

is high, the query is ”active”, and has an high chance to produce relevant dot-product values in the attention computation.

The use of KL divergence as the similarity metric is however computationally expensive, and for this reason a simpler, more efficient query score function $M(q_i, K)$ can be introduced:

$$M(q_i, K) = \max_J \left(\frac{q_i k_J^T}{\sqrt{d}} \right) - \frac{1}{L_K} \sum_{J=1}^{L_K} \frac{q_i k_J^T}{\sqrt{d}} \tag{A.7}$$

This *max-mean measurement* computes the distance between the distribution peak and its mean value: recalling Fig.A.2, high peaks are associated to strong query-key activations, and thus this metric can be effectively adopted to rank queries in order of importance.

Bibliography

[1] J. Alammar. The Illustrated Transformer. URL: https://jalammar.github.io/illustrated-transformer/.

[2] D. Alikaniotis and V. Raheja. The unreasonable effectiveness of transformer language models in grammatical error correction, 2019. arXiv: 1906.01733 [cs.CL].

[3] I. Beltagy, M. E. Peters, and A. Cohan. Longformer: the long-document transformer, 2020. arXiv: 2004.05150 [cs.CL].

[4] J. Brownlee. How to Decompose Time Series Data into Trend and Seasonality. URL: https://machinelearningmastery.com/decompose-time-series-data-trend-seasonality/.

[5] R. Child, S. Gray, A. Radford, and I. Sutskever. Generating long sequences with sparse transformers, 2019. arXiv: 1904.10509 [cs.LG].

[6] K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Davis, D. Belanger, L. Colwell, and A. Weller. Masked language modeling for proteins via linearly scalable long-context transformers, 2020. arXiv: 2006.03555 [cs.LG].

[7] J. Connor, R. Martin, and L. Atlas. Recurrent neural networks and robust time series prediction. *IEEE Transactions on Neural Networks*, 5(2):240–254, 1994. DOI: 10.1109/72.279188.

[8] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov. Transformer-xl: attentive language models beyond a fixed-length context, 2019. arXiv: 1901.02860 [cs.LG].

[9] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: transformers for image recognition at scale, 2021. arXiv: 2010.11929 [cs.CV].

[10] S. Du. Understanding Deep Self-attention Mechanism in Convolution Neural Networks. URL: https://medium.com/ai-salon/understanding-deep-self-attention-mechanism-in-convolution-neural-networks-e8f9c01cb251/.

[11] A. Fan, T. Lavril, E. Grave, A. Joulin, and S. Sukhbaatar. Addressing some limitations of transformers with feedback memory, 2021. arXiv: 2002.09402 [cs.LG].

[12] X. S. Ganchao Bao Yuan Wei and H. Zhang. Double attention recurrent convolution neural network for answer selection. *Royal Society Open Science*, 7, May 2020. URL: https://doi.org/10.1098/rsos.191517.

[13] K. Hatalis. Probabilistic Forecasting: Learning Uncertainty. URL: https://www.analytikus.com/post/2018/03/21/probabilistic-forecasting-learning-uncertainty.

[14] J. Ho, N. Kalchbrenner, D. Weissenborn, and T. Salimans. Axial attention in multidimensional transformers, 2019. arXiv: 1912.12180 [cs.CV].

[15] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, I. Simon, C. Hawthorne, N. M. Shazeer, A. M. Dai, M. D. Hoffman, M. Dinculescu, and D. Eck. Music transformer: generating music with long-term structure. In *ICLR*, 2019.

[16] R. J. Hyndman and G. Athanasopoulos. Forecasting: Principles and Practice. URL: https://otexts.com/fpp3/.

[17] S. M. Kazemi, R. Goel, S. Eghbali, J. Ramanan, J. Sahota, S. Thakur, S. Wu, C. Smyth, P. Poupart, and M. Brubaker. Time2vec: learning a vector representation of time, 2019. arXiv: 1907.05321 [cs.LG].

[18] A. Kazemnejad. Transformer Architecture: The Positional Encoding. URL: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/.

[19] N. Kitaev, Ł. Kaiser, and A. Levskaya. Reformer: the efficient transformer, 2020. arXiv: 2001.04451 [cs.LG].

[20] J. Klaas. *Machine Learning for Finance*. Packt Publishing, Birmingham, UK, 1st edition, 2019. ISBN: 9781789136364.

[21] I. Koprinska, D. Wu, and Z. Wang. Convolutional neural networks for energy time series forecasting. In *2018 international joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2018.

[22] J. Lee, Y. Lee, J. Kim, A. R. Kosiorek, S. Choi, and Y. W. Teh. Set transformer: a framework for attention-based permutation-invariant neural networks, 2019. arXiv: 1810.00825 [cs.LG].

[23] E. Lewinson. *Python For Finance Cookbook*. Packt Publishing, Birmingham, UK, 1st edition, 2020. ISBN: 9781789618518.

[24] S. Li, X. Jin, Y. Xuan, X. Zhou, W. Chen, Y.-X. Wang, and X. Yan. Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting, 2020. arXiv: 1907.00235 [cs.LG].

[25] B. Lim, S. Ö. Arık, N. Loeff, and T. Pfister. Temporal Fusion Transformers for interpretable multi-horizon time series forecasting. *International Journal of Forecasting*, 37(4):1748–1764, 2021.

[26] B. Lim and S. Zohren. Time-series forecasting with deep learning: a survey. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 379(2194):20200209, February 2021. ISSN: 1471-2962. DOI: 10.1098/rsta.2020.0209. URL: http://dx.doi.org/10.1098/rsta.2020.0209.

[27] T. Lin, Y. Wang, X. Liu, and X. Qiu. A survey of transformers, 2021. arXiv: 2106.04554 [cs.LG].

[28] Y. Lin, I. Koprinska, and M. Rana. *Springnet: transformer and spring dtw for time series forecasting*. In November 2020, pages 616–628. ISBN: 978-3-030-63835-1. DOI: 10.1007/978-3-030-63836-8_51.

[29] P. J. Liu, M. Saleh, E. Pot, B. Goodrich, R. Sepassi, L. Kaiser, and N. Shazeer. Generating wikipedia by summarizing long sequences, 2018. arXiv: 1801.10198 [cs.CL].

[30] A. Pappalardo. Exploring the boundary between accuracy and performances in recurrent neural networks. URL: https://necst.it/exploring-boundary-accuracy-performances-recurrent-neural-networks/.

[31] N. Parmar, A. Vaswani, J. Uszkoreit, Ł. Kaiser, N. Shazeer, A. Ku, and D. Tran. Image transformer, 2018. arXiv: 1802.05751 [cs.CV].

[32] M. Pipattanasomporn, G. Chitalia, J. Songsiri, C. Aswakul, W. Pora, S. Suwankawin, K. Audomvongseree, and N. Hoonchareon. Cu-bems, smart building electricity consumption and indoor environmental sensor datasets. *Scientific Data*, 7(1):241, July 2020. ISSN: 2052-4463. DOI: 10.1038/s41597-020-00582-3. URL: https://doi.org/10.1038/s41597-020-00582-3.

[33] J. Qiu, H. Ma, O. Levy, S. W.-t. Yih, S. Wang, and J. Tang. Blockwise self-attention for long document understanding, 2020. arXiv: 1911.02972 [cs.CL].

[34] A. Roy, M. Saffar, A. Vaswani, and D. Grangier. Efficient content-based sparse attention with routing transformers, 2020. arXiv: 2003.05997 [cs.LG].

[35] Y. Sakurai, C. Faloutsos, and M. Yamamuro. Stream monitoring under the time warping distance. In pages 1046–1055, April 2007. DOI: 10.1109/ICDE.2007.368963.

[36] J. Schmitz. Stock predictions with state-of-the-art Transformer and Time Embeddings. URL: https://towardsdatascience.com/stock-predictions-with-state-of-the-art-transformer-and-time-embeddings-3a4485237de6.

[37] P. Shaw, J. Uszkoreit, and A. Vaswani. Self-attention with relative position representations, 2018. arXiv: 1803.02155 [cs.CL].

[38] Y. Tay, D. Bahri, L. Yang, D. Metzler, and D.-C. Juan. Sparse sinkhorn attention, 2020. arXiv: 2002.11296 [cs.LG].

[39] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler. Efficient transformers: a survey. *CoRR*, abs/2009.06732, 2020. arXiv: 2009.06732. URL: https://arxiv.org/abs/2009.06732.

[40] C. Toth, P. Bonnier, and H. Oberhauser. Seq2tens: an efficient representation of sequences by low-rank tensor projections. In *International Conference on Learning Representations*, 2021. URL: https://openreview.net/forum?id=dx4b7lm8jMM.

[41] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jegou. Training data-efficient image transformers & distillation through attention. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 10347–10357. PMLR, July 2021. URL: https://proceedings.mlr.press/v139/touvron21a.html.

[42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[43] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma. Linformer: self-attention with linear complexity, 2020. arXiv: 2006.04768 [cs.LG].

[44] N. Wu, B. Green, X. Ben, and S. O'Banion. Deep transformer models for time series forecasting: the influenza prevalence case. *ArXiv*, abs/2001.08317, 2020.

[45] Y. Xu, H. Wei, M. Lin, Y. deng, K. Sheng, M. Zhang, F. Tang, W. Dong, F. Huang, and C. Xu. Transformers in computational visual media: a survey. *Computational Visual Media*, 8:33–62, October 2021. DOI: 10.1007/s41095-021-0247-3.

[46] G. Zerveas, S. Jayaraman, D. Patel, A. Bhamidipaty, and C. Eickhoff. A transformer-based framework for multivariate time series representation learning. In New York, NY, USA. Association for Computing Machinery, 2021. ISBN: 9781450383325. DOI: 10.1145/3447548.3467401.

[47] H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, and W. Zhang. Informer: beyond efficient transformer for long sequence time-series forecasting. In *The Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021*, online. AAAI Press, 2021.