POLITECNICO DI TORINO

Master's Degree in Data science and engineering

Master's Degree Thesis

Transformer attention optimization in time series forecasting

Supervisors

Prof. Francesco VACCARINO

Co-supervisors

Rosalia TATANO

Candidate

Andrea ARCIDIACONO

April 2022

Abstract

Transformer-based architectures are neural networks architectures developed for natural language processing. These state-of-the-art architectures innovation is the use of the self-attention mechanism. These models have been deployed in several settings, not just limited to natural language, but also including videos and images. However they are hard to scale up for industrial applications due to the quadratic time and memory complexity of attention mechanism. Therefore, there has been a extensive research in proposing new variants of these architectures to solve this problem approximating the quadratic cost attention matrix, making the model more efficient and more lightweight. This thesis is focused on analyzing the recently proposed efficient attention mechanisms of Performer, BigBird and Informer and apply them to the task of time series forecasting. In particular, starting from the implementation of the Informer, the attention mechanisms of Performer and BigBird are integrated in its architecture, resulting in four models to be tested: Informer with vanilla attention mechanism, Informer with the so-called ProbSparse attention, Informer+Performer, Informer+BigBird. We compere the performance of each variation, in a easy to access scalable hardware, in two real industrial problems focusing on performance versus resources needed. The results suggest that the accuracy of the forecasting is similar for all tested models, but the computational perfomance on resources varies substantially. In fact, many efficient architecture loose the quadratic complexity problem but introduce more complex algorithms to calculate the attention approximation and this increases the overhead memory requirement. Thus, the obtained results suggest that any efficient Transformer architecture modification has to be carefully chosen according to the task and dataset characteristics, as the use of these efficient models might not result in significant benefits in terms of computational resources needed for training and testing of such models.

Acknowledgements

ACKNOWLEDGMENTS

To all that were involved into this hard work, and to all that were just supporting, thank you.

Table of Contents

List of Tables

List of Figures

Introduction

The Transformer [1] architectures are deep neural networks originally developed for natural language processing. They are designed to resolve the problems faced by RNN [2] architecture using the self-attention mechanism, a similarity feature score, core of the innovation of the Transformer. We can now find Transformers models in many of the big tech companies algorithms, not only just limited to natural language tasks, but also including videos, images and time series tasks.

In this thesis, done in collaboration with *AddFor Industriale S.r.l.*, we will focus on Transformer architectures for time series forecasting. Transformer architectures are extremely powerful but have some important complexity drawbacks that make the architecture hard to scale up. In fact the self-attention present a quadratic computational complexity with respect to the input length that limits the use case of the architecture only to low input length. For this reason, most of the recent research on Transformers is focused on developing an approximation of the attention mechanism to reduce the complexity.

In particular, we focus on the recently proposed efficient attention mechanisms of Performer [3], BigBird [4] and Informer [5] and apply them to the task of time series forecasting on publicly available datasets. These three attention represent 3 different ways of approximation, respectively utilizing "learnable pattern", "fixed pattern" and "kernel approximation". The Informer architecture functions as the back bone for our work and all the other attention mechanisms are integrated in its architecture, resulting in a total four models to be tested: Informer with vanilla attention mechanism, Informer with the so-called ProbSparse attention, Informer+Performer, Informer+BigBird.

This thesis is organized as follows. In chapter 1, we introduce what a time series dataset is and how to manage it for a forecasting task. We further introduce in chapter 2 the main concept of deep learning, what a neural network is, how to train it and what are the main building block utilized to develop a model. In the end of the second chapter we also discuss the first Transformer architecture [1]. In chapter 3 we discuss the complexity problem of the Transformer architecture and we present the results of an extensive research into the possible solution of the complexity problem, presenting a taxonomy to organize the state of the art. The efficient architectures that are the focus of this thesis are described in chapter 3, where we provide a theoretical comparison between the various efficient attentions. In chapter 4 we present briefly our testing bench and all the process involved in designing the experiments before focusing on the analysis of the results.

Chapter 1

Time Series Forecasting

In this chapter we aim to introduce what the time series are, how they are defined and what are the main application regarding this type of data. In particular we will see the difference between multivariate and univariate time series, and how the time series can be decomposed. Moreover we will talk about linear regression and non linear regression applied to time series. We will also introduce the concept of time series forecasting, the main task in this thesis, and apply it for both univariate and multivariate time series. We will also talk briefly about the statistical model used in forecasting and the deep learning models.

1.1 Time Series

A Time series is a collection of points drawn at successive equally spaced points in time (e.g., hourly, daily, weekly, monthly, quarterly, annually), so discrete in time. They are widely used datasets as any real world measurement provided in time can be considered a time series, for examples the temperature in a room or the number of patients in a hospital. A continuous function can also be considered a time series if we discretize its values using a sampling frequency, for example acoustic data are time series as digital audio file have a frequency of sampling so it matches perfectly the time series definition.

We can visualize time series data using the time plots [6]. That is, data points are plotted against the time of observation, with consecutive points joined by straight lines. In figure 1.1 we can see and example of time plot for a time series.

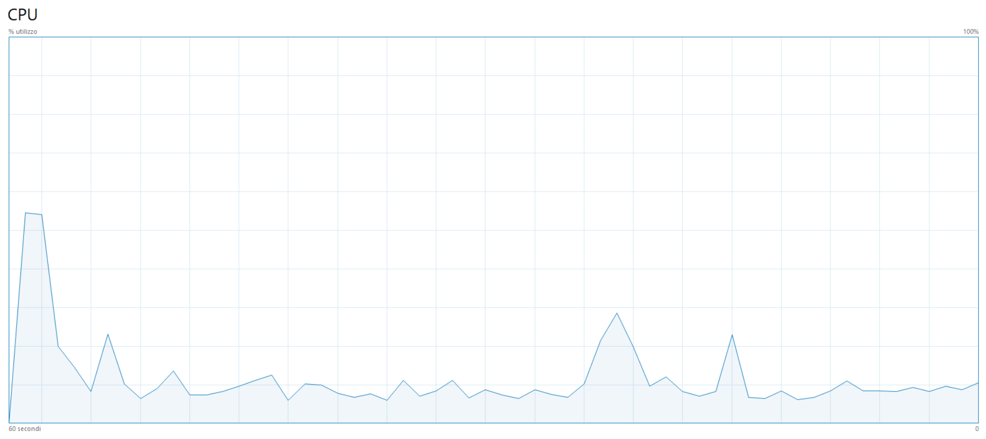

Figure 1.1: % of usage of Windows CPU over 60 seconds

We can analyze and drawn meaningful statistics from a times series, but for the most part this datasets are used in time series forecasting and regression. We will go into details of each task in Sec. 1.3 and 1.2. Times series are also used in anomaly detection task.

We can distinguish between two type of time series datasets, multivariate and univariate. The former contains multiple features, or variables, that are correlated and can influence one another, the latter presents a single time-dependent variable.

1.1.1 Univariate series

The simplest case of time series dataset is univariate time series with a single time-dependent variable. We provide here a definition of time series as:

$$\bar{x} = (x_1, x_2, x_3, ...) \tag{1.1}$$

As we can see from definition, the time series doesn't need to have any time stamp data, it's just a series of scalars, this is because as time series points are equally spaced in time, time stamps doesn't add any information. In any case time step information is used only for visualization purposes.

1.1.2 Multivariate series

Multivariate time series datasets consists of multiple time series in a single dataset. The single time series share time steps and are correlated. An example of time series for weather is provided in Table 1.1.

time	temperature	cloud coverage	humidity
5:00	22°C	95%	60%
6:00	23°C	75%	73%
7:00	25°C	62%	82%
8:00	27°C	60%	75%
9:00	28°C	40%	68%
10:00	30°C	33%	66%

Table 1.1: Multivariate time series of weather attributes

The dataset depicted in Table 1.1 presents three variables. Usually in performing regression or forecasting tasks only a variable is selected as target. The target is the only variable responsible for the performance of the task but regression/forecasting algorithms are applied to all variables. In addition to auto correlation information in the multivariate settings we can exploit also the correlation between variables to achieve better results.

1.1.3 Trend and seasonality in time series

Time series can have different patters, and it is helpful to divide each component following the philosophy of DIVIDE ET IMPERA.

When decomposing a time series, it is sometimes helpful to first transform or adjust the series in order to make the decomposition (and later analysis) as simple as possible. In "Forecasting: Principles and Practice"[7] the authors provide us some example for transformation related to particular type of time series. For example we adjust monthly time series because the number of days of a month vary, or we use adjust for inflation in economics analysis as the value of a currency change over time.

The most general time series can be considered the result of three different components: seasonal (S), trend-cycle (T) and leftovers (R) [7]. A trend (T) is a long-term increase or decrease in the data. It does not have to be linear.

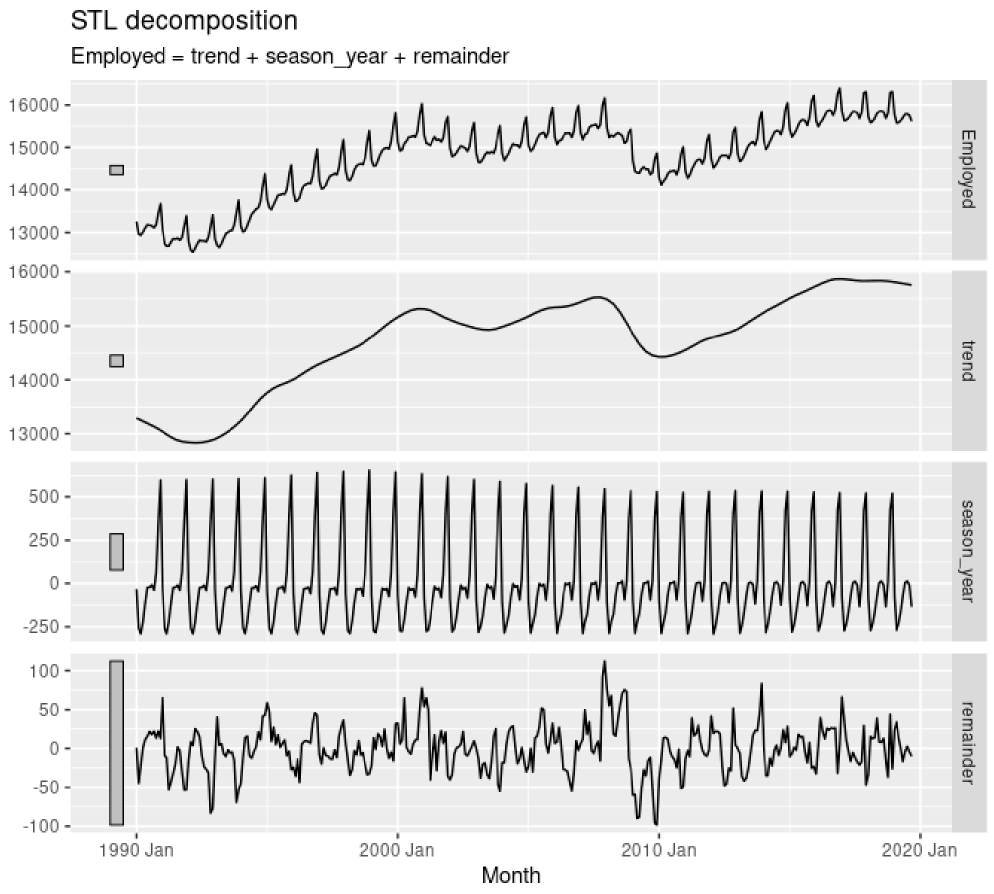

Figure 1.2: Total US retail employment (top) and its three additive components obtained from a robust STL decomposition with flexible trend-cycle and fixed seasonality[7].

A cycle (T) occurs when the data exhibit rises and falls that are not of a fixed frequency.
A seasonal pattern (S) occurs when a time series is subject to seasonal factors such time of the year or day of the week. Seasonality is always of a fixed and known period.

For example we can think of the sales of a store, they have an upward trend and maybe a periodic seasonal pattern where in the holiday season or sale season they increase. They also have weekly pattern maximum sale in the weekends and maybe aperiodic patter for particular national or local events. Further they present a random variation (R) that can be considered as normally distributed.
Time series can be modeled with additive or multiplicative decomposition (1.2).

Additive decomposition is often used when the amplitude of the series do not vary in time, otherwise we use multiplicative decomposition. These additive and multiplicative decomposition for a time series X are defined respectively as:

$$\bar{x} = S + T + R \ , \quad \bar{x} = S \times T \times R \tag{1.2}$$

We can think of multiplicative decomposition as a additive decomposition after a logarithmic transformation as for the property of logarithms holds that :

$$\log(S \times T \times R) = \log(S) + \log(T) + \log(R) \tag{1.3}$$

In Figure 1.2, we can see an example of decomposition of a time series following the STL (Seasonal and Trend decomposition using Loess) method [8]. The first graph is the total time series, while the three graph underneath are respectively the T, S and R components of the time series.

We use decomposition to understand more about the underling patters of the time series, but it can also be used to improve forecast accuracy. In particular, if for the analysis that we will perform the seasonality component is considered just noise, we can remove it before hand to reduce the analysis error. In this case we are using “seasonally adjusted” data.

1.2 Regression

As we have already touch upon autoregression model in this section we will go in depth about what actually means regression.
Regression is the art of estimating the relationships between a dependent variable (often called the ’outcome’ or ’target’ variable) and one or more independent variables (feature, predictors). The aim is to find a function f, also called model, and parameters $\bar{\beta}$ to model the output variable y_i minimizing the error ϵ_i, i.e.

$$y_i = f(\bar{x_i}, \bar{\beta}) + \epsilon_i \tag{1.4}$$

where $\bar{x}_i$, represent the predictors vector values at step i. Above equation is the general formulation [9] of the regression problem.

1.2.1 Linear Regression

The easiest form of regression is linear regression, where we try to find a hyperplane that minimizes some error function or loss function. In linear regression f is simply a linear function. If we use a simple matrix multiplication $\bar{x_i}^T\bar{\beta}$ Eq. (1.4) can be rewritten as:

$$Y_i = \bar{X_i}^T\bar{\beta} + \epsilon_i \tag{1.5}$$

Note that regression can be applied also to sequential data that are not time series. In Figure 1.3 we can see a set of points using a scatter plot visualization. The red line is the output of a linear regression model trained on blue points. In this case, we have a single predictor and a single target variable, but in general multivariate regression, or multiple regression model, is possible as X in Eq. (1.5) is a matrix of dimension (n_sample, n_predictors).

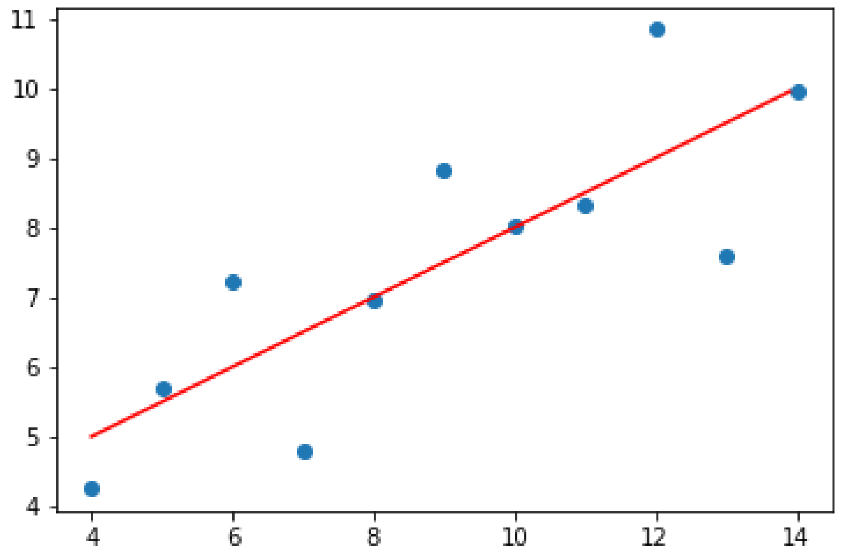

Figure 1.3: Scatter plot of points (blue) with regression (red)

When we use a linear regression model, we are implicitly making some assumptions about the variables in Eq.(1.5). First, we assume that the problem can be modelled by a linear equation, secondly we make the following assumption about errors random variable ϵ:

- They have zero mean, typically normal distribute.
- They are not autocorrelated.
- They are unrelated to the predictor variables.

This assumption holds if there is no more information to be exploited in the leftover term R. Another useful assumption is that variable predictor X are not random variable, this assumption is often let go as it is practically impossible with real world data.

We aim to find parameters β to estimate our function. A standard method to find the hyperplane in linear regression is the least square method that can be written as:

$$\min_{\epsilon} \sum_i \hat{\epsilon^2} = \min_{\hat{\bar{\beta}}} \sum_i y_i^2 - (\bar{x_i}^T \hat{\bar{\beta}})^2 \tag{1.6}$$

where ϵ are the errors, β is the parameter vector of our model, y_i are the value of the target variable and $\bar{x_i}$ are predictors values. This is called least squares estimation because it gives the least value for the sum of squared errors. As our aim is to find parameters β and solving for β we get:

$$\hat{\bar{\beta}} = \arg\min_{\bar{\beta}} ||\bar{y} - \mathbf{X}^T \bar{\beta}||^2 \tag{1.7}$$

Finding β is called fitting the model to the data, as our model (the linear regressor) depends on the data.

One assumption in the least square method is that predictors $\mathbf{X}$ are independent, but in many instances this is not the case. If this happens it could lead in to increase error. We can add some regularization term to account for this phenomenon but this is beyond the scope of this introduction.

Given the β coefficients we now have a complete model that we can query, so we can now evaluate our model. The differences between the observed y values and the corresponding fitted $\hat{y}$ values are called residuals and are defined as:

$$e_i = y_i - \hat{y}_i \tag{1.8}$$

Each residual is the unpredictable component of the associated observation. In some cases in real world data auto correlation in error is present as the observation taken close in time are also close in value, this violate the second assumption in Figure 1.2.1 as more information can be extrapolated. This is still an unbiased estimator but it has high variance and so slow convergence. In some cases is better to have a biased estimator with low variance.

The detection of outliers is another step that we can apply to improve the model. Outliers are influential observation that do not follow the distribution of data and that steer the model away from the desired configuration. Following equation (1.6) if some points have a high value of error are more influential as model tries to optimize every point. In this case we can deploy some of the mechanism to detect outliers and remove them from the training set. Outliers can be produced by sensor failure or mistyping[10].

Another way to increase out model performance is with data augmentation. In this case a possible strategy if the time series dataset has seasonal cycles we can use a dummy variable to account for holidays or weekends. For example let imagine a dataset depicting the electric consumption of a city. In this case industries of the city will be closed during weekends or holidays so we can add a **True** or **False** variable accounting for these events. The dummy variable becomes another predictor x in our regression analysis[11].

1.2.2 Non linear Regression

Note that for linear in linear regression we means linear with respect to parameters β, this means that even non linear relationship with predictors can be modelled. For example we imagine a quadratic relation. $y = 3x + x^2$ the β parameter in this case we have defined them as 3 and 1.

We can see the time series y with respect to both predictors x and x^2 in figure Figure 1.4. In this case the second assumption in Figure 1.2.1 is violated, but the only results will be that the parameters of the model will be slightly off with respect to ours. So for usage outside the training set this model will perform poorly.

Using linear regressor of sci-kit learn we can find the following parameter values:

```
[3.73655067, 0.9462789 ]
```

and we can plot the prediction of our model vs the real curve Figure 1.5. We can see that the model performs well within the training domain, even if parameters vaule are slightly off

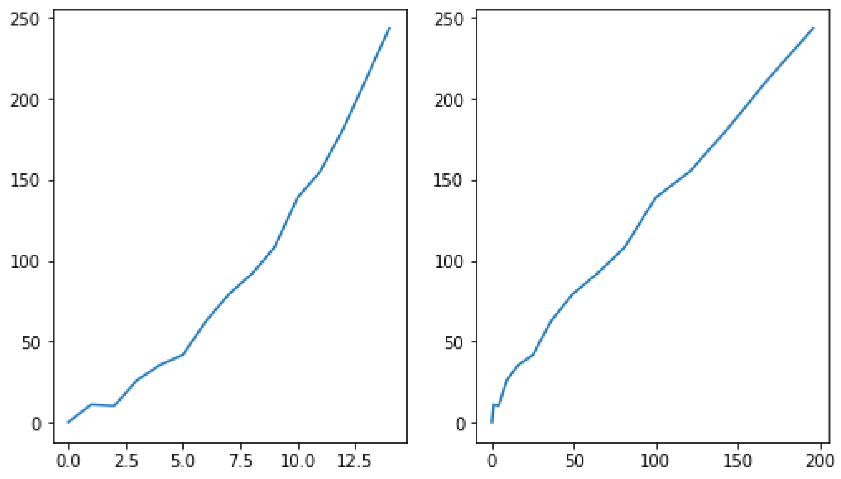

Figure 1.4: Time series of $y = 3x + x^2$ plotted with the two predictors x (right) x^2 (left)

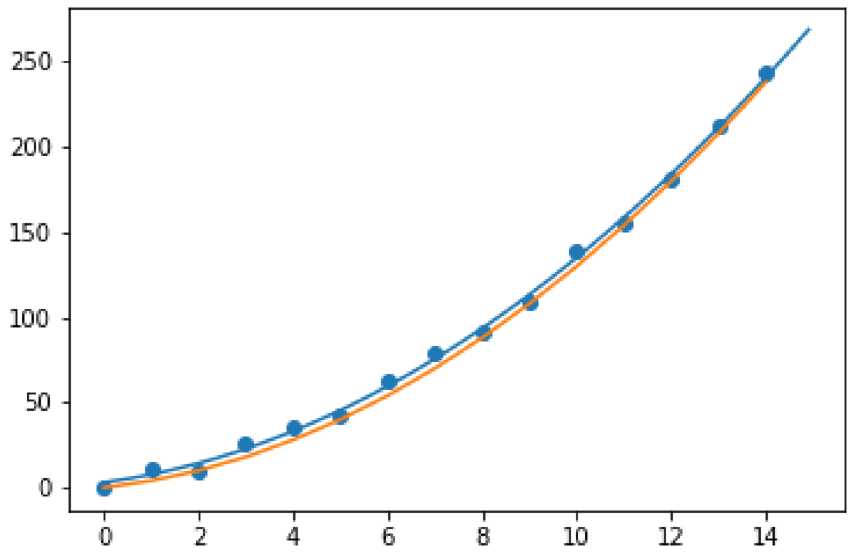

Figure 1.5: Solution of our model in orange, target value and time series in blue

In actual non linear regression a non linear function is applied to y so we can resolve the equation we regain a similar form as (1.4) where f is non linear and also parameters β are subject to f.

1.2.3 Autoregression

Autoregressive models differs from standard regressive models in that they use past value of the variable as predictors, so the terms auto mean regression to itself [12]. So an autoregressive model can be written as:

$$y_i = f(\bar{y}, \bar{\beta}) + \epsilon_i \tag{1.9}$$

This concept is really power full as for most measure the past values of it-self carry a lot of information. We can think of a temperature in a room, if the time step is relatively short, surely the value of the temperature a time i is correlated with past values. An important choice here is the windows for the autoregression, so we take only the value close in time, or also further? In this case if we know "a priori" that there is a seasonal component we can exploit this knowledge taking also values according to the seasonal frequency.

1.3 Forecasting

But what if instead of predicting only in the training domain we try to extend our regression model will to future points? We would enter the realm of forecasting. The art of forecasting has been used throughout history, we started with magicians and rituals to try and predict or influence weather events, people's actions or crop's growth rate. Our society now uses predictions for almost anything, they are so important because they provide us a way to plan in advance.

It will come as no surprise that the most common type of data in time series is often subject to forecasting and it is applied to a variety of industries. In forecasting for time series there are some important concept we must introduce, first we have the forecasting windows. The lengths of this windows l is defined as how many points we want to predict. We can then do a dynamic forecasting or a single sweep forecasting. This actually depends on the model architecture but dynamic forecasting predicts one point at a time, and uses the new and updated time series for forecasting the next one. Regression model are preferably used as forecasting models in time series, this is because it's an easy extension of the concept.

1.3.1 Univariate Forecasting

As we have discussed previously univariate forecasting uses only one time series, for this reason all the model used in this area are purely autoregressive models. Imagine a autoregressive model ($\hat{f}$) with dynamic inference that outputs one point and take, without loss of generality, as input the previous $n-1$ points in the time series. We can write the equation as:

$$y_i = \hat{f}(y_j, \bar{\beta}) + \epsilon_i, \quad j \in \{i-n, i-1\}, \quad n < i \tag{1.10}$$

This inference is repeated l time as l is the length of the prediction window, so this techniques is recursive as output of the model are included in the input at the next time step computation. While dynamic inference can be good because l becomes a single hyperparameter of the model, and varying it doesn't require any adjustment, can be also bad as error propagation can become quickly unsustainable for high values of l. A solution can be single sweep forecasting, that compute all the forecasting window in one single step. This has the drawback that the model is tailored for that specific window length.

1.3.2 Multivariate Forecasting

Multivariate forecasting uses all the tools and applied them to all variable in the dataset, both predictors and target. But we got extra also the tools of regression between the predictors and the target variable. All the previous concept of univariate forecasting and regression still applies for multivariate regression.

1.3.3 Statistical models

As we have seen previously the first type of model used in for time series forecasting where purely autoregressive, nowadays the state of the art for time of purely statistical models in time series forecasting are the ARIMA model. ARIMA strand for Auto Regressive Integrated Moving Average, its aim to describe the autocorrelations in the data. We have already discussed the AR part of this model, we will now discuss the remaining 2 parts.

Rather than using past values of the forecast variable in a regression, a moving average model uses past forecast errors in a regression-like model [12].

$$y_i = \hat{f}(e_j, \bar{\theta}) + e_i, \quad j \in \{i-n, i-1\}, \quad n < i \tag{1.11}$$

In this equation e are the forecast errors and e_i is just white noise as we cannot really observe it before computing our estimate on y_i.

It can be demonstrated that Moving Average and Autoregressive models are closely link in that is possible to write ant stationary AR model as a MA model. We will omit the proof that can be found in "Forecasting: Principles and Practice" [12].

ARIMA models works only with stationary time series so time series that do not change over time. One technique to make non stationary time series into stationary is using differentiation. It involves looking only at the differences between consecutive observation. Differencing can help stabilise the mean of a time series by removing changes in the level of a time series, and therefore eliminating (or reducing) trend and seasonality[12]. In fact if trend or seasonality are present in the time series this is non stationary.

Differenced series can be defined as:

$$y'_i = y_i - y_{i-1} \tag{1.12}$$

The differentiated series start form the second value as no difference can be defined for the first. Differencing is the I part in the ARIMA, in this case “integration” is the reverse of differencing. So the full ARIMA model combining equation (1.9), (1.11) and (1.12) can be written as:

$$y'_i = \hat{f}(y'_k, \bar{\beta}; e_j, \bar{\theta}) + e_i, \quad k \in \{i-n, i-1\}, j \in \{i-p, i-1\} \tag{1.13}$$

With both $n < i; p < i$ where n and p are the two window lengths for autoregressive and moving average.

This model is also called the non-seasonal ARIMA [12], they typically use maximum likelihood estimator to find the values of both parameters $\bar{\beta}$ and $\bar{\theta}$, this is similar to the least squared method we discussed in section 1.2.

There exist also the Sasonal ARIMA models that extends the capabilities of the model. The model adds an additional seasonal term that are simply multiplied by the non-seasonal terms. The seasonal part of the model consists of terms that are similar to the non-seasonal components of the model, but involve backshifts (going back in time) of the seasonal period.

It is always useful to add prediction interval to forecast predictions. Prediction interval are defined as $I : P(y_i \in I) \geq 1 - \alpha$ where I is the interval and α is the confidence level. So in this case if $\alpha = .05$ we are 95% sure that the real value y_i will be inside our interval I. This is not a trivial calculation, more detailed can be found in Introduction to Time Series and Forecasting by Peter J. Brockwell and Richard A. Davis [13], but ARIMA model can output also this information.

Chapter 2

Deep Learning

While classical statistical model are really powerful and can be easily explained, the state of the art for time series forecasting are the deep learning models. In this chapter we will talk in detailed about the principles of deep learning and some of the architecture that are used in time series forecasting.

Deep learning is a particular field of study inside machine learning, also called patter recognition. It studies in particular neural networks (NN), it is called "deep" because of the depth in NN architecture, that are based on the mathematical equation that loosely describe the functioning of a biological neuron.

In this section we will understand the basic algorithm and principles of deep learning and we will present some of the architecture that have been applied to time series forecasting.

The key idea behind NN is to mimic a biological neuron in it's simplest mathematical formulation.
Biological neurons are linked to other neuron by various connection. We can see an example of a simple spiking neuron in Figure 2.1, here we can recognize input links as dendrites, and output links as axon terminals. We will consider the output unique in our formulation as it does not provide any generalization challenges. The center of the neuron, the soma, is were the computation takes place. Here the electrical inputs are summed up and confronted with the membrane voltage. If the input voltage exceed the membrane voltage an electrical pulse (spike) is fired to be transmitted to other neurons through terminals. More in depth into the biology of brains and neurons can be found in "Principle of Nerual Scienze" by Eric R. Kandel [14].

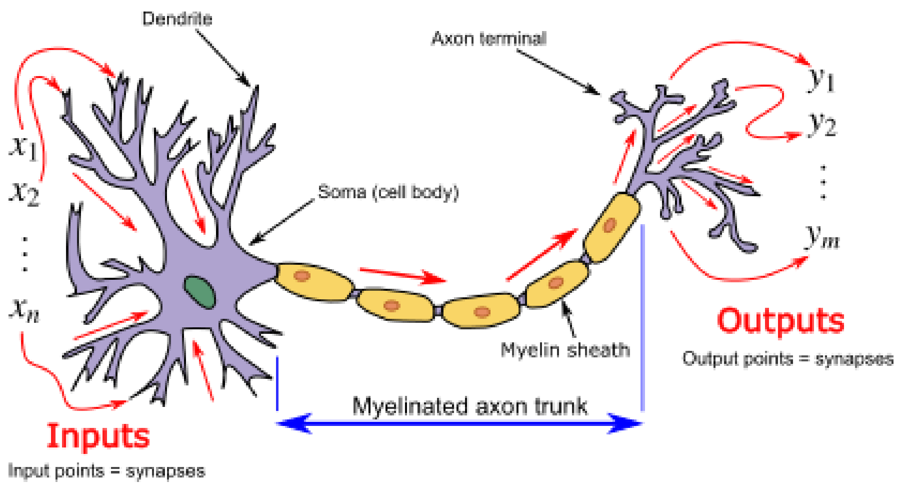

Figure 2.1: Neuron and myelinated axon, with signal flow from inputs at dendrites to outputs at axon terminals. The signal is a short electrical pulse called action potential or 'spike' [15].

Numerous formulation have been proposed to model the neuron function, some are more biologically accurate then others, but we are not interest in biological accuracy so we will present the linear threshold function formulation:

$$y = \phi\left(\bar{x}\right) \tag{2.1}$$

with

$$\phi(\bar{x}) = \begin{cases} 1, & \text{if } \bar{w}\bar{x} + \theta > 0 \text{ with } x_i \in \{-1{,}1\} \\ -1, & \text{otherwise} \end{cases} \tag{2.2}$$

where θ is our threshold and models the membrane potential, $\bar{x} = (x_1, x_2, \ldots, x_i, \ldots, x_L)$ is the input vector and $\bar{w}$ is a weight vector.

This is the basis for Neural network computation, in this case we are using a sign function ϕ to model the neuron, ϕ is also called activation function. In general there are other activation functions used that perform better, some of them can be found in Table 2.2. We will understand pro and cons of each function in subsection 2.1.3

2.1 Perceptron

Eq. (2.2) describes the biological neuron, but a single neuron has only two states, fired or not fired. This is a binary output and means that single neuron can only perform classification. The binary classification algorithm that model a single neuron is called the perceptron [16].
Classification is the task to assign at each data point $\bar{x}$ a data class d. Classification is a sub problem of regression as classification can be achieved with a logistic regressor. For this reason it has been developed an extension to the perceptron that addresses regression problems. We will see the regression formulation later in this chapter. Mathematically we have seen the perceptron's activation function in equation (2.2), so we want to find a procedure to estimate $\bar{w}$ given a dataset D:

Algorithm 1 Perceptron algorithm

```
▷ $y_i$ is the output of the function $\phi$
▷ $d_i$ is the data class, $y_i, d_i \in \{-1,1\}$
▷ $D$ is the dataset, and $\hat{\omega}$ the weight vector
▷ Initialize $\hat{\omega}$
$\hat{\omega} \leftarrow 0$
$\theta \leftarrow 0$
for $\bar{x_i}, d_i \in D$ do
    $y_i(t) = \phi(\bar{x}_i; \hat{\omega}, \theta) = sign(\hat{\omega}\bar{x}_i + \theta)$
    if $d_i \cdot y_i \leq 0$ then
        $\hat{\omega} \leftarrow \hat{\omega} + y_i \cdot x_i$
        $\theta \leftarrow \theta + y_i$
    end if
end for
```

It is an online learning algorithm of the class of error driven learning, it means that model's parameters are updated only in case of an error [16]. We can see this characteristic in the algorithms above. The values of θ and $\hat{\omega}$ are updated only if $d_i \cdot y_i \leq 0$, so if an error occur. The perceptron is a linear classifier, so if the dataset is not lineally separable it will fail to find a solution to the problem [16]. This is a real big drawback, but fortunately there is method that let us extend this and other algorithm to model non linear function.

2.1.1 Kernel Trick

Kernel methods are machine learning methods that use kernel functions to project and operate in a high-dimensional, implicit feature space without ever computing the coordinates of the data in that space, but simply computing the inner products between data points [17].
The perceptron is an algorithm compatible with the kernel trick, we can rewrite $\bar{\omega}$ as a linear combination of the inputs $\bar{x_i}$:

$$\bar{\omega} = \textstyle\sum_i \alpha_i d_i \bar{x_i} \tag{2.3}$$

where α_i is the number of times $\bar{x_i}$ was misclassified. Then:

$$y_j = sign(\bar{\omega}^T \bar{x_j}) = sign((\textstyle\sum_i \alpha_i d_i \bar{x_i})^T \bar{x_j}) = sign(\textstyle\sum_i \alpha_i d_i \langle \bar{x_i}, \bar{x_j} \rangle) \tag{2.4}$$

We have reached a formulation of the problem where the output is a function of the scalar product between two points. This scalar product can be swapped for a Kernel function K. A kernel is a symmetric continuous function positive semi-definite:

$$\begin{aligned} &K : \mathcal{X} \times \mathcal{X} \to \mathbb{R} \\ &\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j K(\bar{x_i}, \bar{x_j}) \\ &K(\bar{x_i}, \bar{x_j}) = \langle \Phi(\bar{x_i}), \Phi(\bar{x_j}) \rangle \end{aligned} \tag{2.5}$$

Where $\Phi(\bar{x_i})$ is the implicit mapping of $\bar{x_i}$ into the high dimensional space Hilbert space $\mathcal{X}$.

With this in mind we can swap the inner product in (2.4) with a kernel K and we will get:

$$y_j = sign \textstyle\sum_i \alpha_i d_i K(\bar{x_i}, \bar{x_j}) \tag{2.6}$$

Obviously not every function can be considered a kernel and in principle is not trivial to prove that one function can be considered a kernel, there is a theorem that we can use, the Mercer's theorem. For this formulation we consider $\mathcal{X} = [a, b]$.

Linear kernel	$K(\bar{x_i}, \bar{x_j}) = \bar{x_i}^{\mathrm{T}}\bar{x_j}$
Polynomial kernel	$K(\bar{x_i}, \bar{x_j}) = (\bar{x_i}^{\mathrm{T}}\bar{x_j} + r)^n, \quad \bar{x_i}, \bar{x_j} \in \mathbf{R}^d, r \geq 0, n \geq 1$
Gaussian kernel	$K(\bar{x_i}, \bar{x_j}) = e^{-\frac{\lvert\lvert \bar{x_i} - \bar{x_j} \rvert\rvert^2}{2\sigma^2}}$

Table 2.1: Most used kernels

Theorem 2.1.1 (Mercer's theorem) *Suppose K is a continuous symmetric non-negative definite kernel* (2.5)*. Then there is exist an orthonormal basis $\{e_k\}_k$ of $L^2[a,b]$ continuous on $[a,b]$ and $\{\lambda_k\}_k$ values nonnegative such that K has, with absolute and uniform convergence, the following representation:*

$$K(x_i, x_j) = \sum_{k=1}^{\infty} \lambda_k e_k(x_i) e_k(x_j) \quad \forall (x_i, x_j)$$

In Table 2.1 we provide some of the most used kernels.

The kernel trick is used to leverage the potentiality of a non linear model but retaining the computational efficiency of a linear model [17].

2.1.2 Multi layer perceptron

Returning at the analogy with the nervous system, we can see an artificial neural network (ANN) as a series of perceptron (neurons) that interact with each other as in Figure 2.2.

We can distinguish between three type of neurons.

- Input neurons (blue), that are fixed in number and correspond to the dimension of the input vector
- Output neurons (green), that are fixed in number and correspond to the dimension of the output vector
- Hidden neurons (yellow), that can be any number.

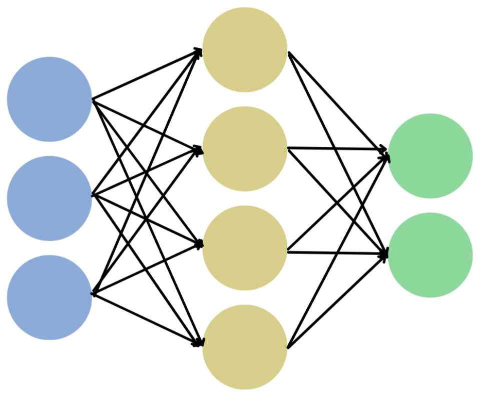
Figure 2.2: Multi layer perceptron or Artificial neural network

sigmoid	$\phi(x) = \frac{1}{1+e^{-x}}$
ReLU	$\phi(x) = \max(0, x)$
leaky ReLU	$\phi(x) = \begin{cases} x, & \text{if } x > 0 \text{ with } x_i \in \{-1{,}1\} \\ 0.01x, & \text{otherwise} \end{cases}$
tanh	$\phi(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$

Table 2.2: Most used activation functions

Both the width and the depth of the hidden layer are hyperparameters of the model, so there can be more hidden layers. If we imagine a network of depth 4, there will be 2 hidden layers.
We have seen that for the perceptron the activation function ϕ was a step function, but in principle we can choose any function according to our task. The most common type of activation functions are depicted in Table 2.2.

Output neurons' activation function is really important as it is defined by our task. For example in classification we will use sigmoid or tanh to have a logistic regressor, while in regression we can leave the output layer with an identity function. Every neuron has its own activation parameters and activation function, even though the latter is usually identical for neuron that belong to the same layer.

We can see in [16] that the perceptron has the capabilities of modelling boolean function **AND** and **OR**, while multilayer perceptron adds the power to model **XOR** function. This leads us to prove that the multilayer percptron is an universal approximator. No formal bounds are given on the number of hidden units required, but accuracy may be increased to the desired value by increasing their number.

2.1.3 Optimization and loss

Training a multilayer perceptron is the same as training a perceptron; the only difference is that now the output is a nonlinear function of the input thanks to the nonlinear basis function in the hidden units. For output neurons we have directly the output value to aim at, but for the other neurons we use the chain rule to calculate gradients.

We provide here first some useful definition first:

- L is the number of layer (the depth)
- $W^l = (w^l_{jk})$ is the weight matrix of layer l
- f^l is the activation function of layer l
- $a^l_k = \bar{w}^l_k \bar{o}^{l-1}$ is the activation value of layer l and neuron k seen in Eq. (2.2)
- $\bar{o}^l$ is the output of layer l

Now we can write the multilayer perceptron as:

$$\bar{y} = g(\bar{x}) := f^L(\mathbf{W}^L f^{L-1}(\mathbf{W}^{L-1} f^{L-2}(\cdots f^1(\mathbf{W}^1 \bar{x})))) \tag{2.7}$$

First we have to introduce the concept of loss function. A loss function is a function that maps an event or values of one or more variables onto a real number intuitively representing some cost, or error, associated with the event. An optimization problem seeks to minimize a loss function [18]. Given a loss function E and the target variable d we can calculate the gradient of the loss E with respect to weight value w^l_{jk} with the gradient chain rule as:

$$\frac{\partial E(d,y)}{\partial w^l_{jk}} = \frac{\partial E}{\partial a^l_k}\frac{\partial a^l_k}{\partial w^l_{jk}} = \frac{\partial E}{\partial \bar{a}^L}\frac{\partial \bar{a}^L}{\partial \mathbf{W}^L}\frac{\partial \mathbf{W}^L}{\partial \bar{a}^{L-1}}\frac{\partial \bar{a}^{L-1}}{\partial \mathbf{W}^{L-1}}\cdots\frac{\partial \mathbf{W}^{l+1}}{\partial a^l_k}\frac{\partial a^l_k}{\partial w^l_{jk}} \tag{2.8}$$

This is called back propagation as we can backpropagate the error (output of the function E) to every parameter of the model to update it, to update is used a technique called gradient descend. The gradient descent procedure minimize E starts from a random $\bar{w}_k^l$, and at each step, updates $\bar{w}_k^l$, in the opposite direction of the gradient. This procedure is repeated for every $\bar{w}_k^l$ in the architecture.

$$\nabla_{\bar{w}_k^l} E = \left[\frac{\partial E}{\partial w_{1k}^l}, \frac{\partial E}{\partial w_{2k}^l}, \frac{\partial E}{\partial w_{3k}^l}, \dots \right] \tag{2.9}$$

The update rule is:

$$\begin{gathered} \Delta w_{jk}^l = -\eta \frac{\partial E}{\partial w_{jk}^l}, \forall j \\ w_{jk}^l \leftarrow w_{jk}^l + \Delta w_{jk}^l \end{gathered} \tag{2.10}$$

Where η is called the stepsize, or learning factor, and determines how much to move in that direction [19]. We recall that we reach a minimum when the derivative is 0. So the procedure finds the nearest minimum that can be also a local minimum, and there is no guarantee of finding the global minimum unless the function has only one minimum. η also has a key role; if it is too small, the convergence may be too slow, and a large value can lead to oscillations or even divergence.
Nearly all of deep learning is powered by one very important algorithm: stochastic gradient descent that is an extension of the gradient descent algorithm. The difference here is that gradient descent is applied to all dataset, but this is not feasible for large datasets as computational complexity is quadratic with respect to dataset size. So stochastic gradient descent introduces batches, it divide the dataset into small chunks and train the model for each chuck separately. Elements in batches are drawn uniformly from the training set [20]. It is called stochastic because we do not compute the full gradient but compute only the gradient on the mini batches and update after every batch, so it depends on the distribution of batches.

In more complex optimizer algorithms, such as ADAM [21], there can be other parameters as well. ADAM for example adds also the possibility to reduce step size if the loss doesn't descent for a fixed amount of iteration, or momentum to increase the value of step size.

We have seen in Eq. (2.10) that weight updates depends on the gradient (2.9) that depends on function topology. In particular there can be present some cliffs that leads to a high value of ∇E and consequently an high value for the update. This is undesirably as an update step can move the parameters extremely far, so we can implement gradient clipping [18]. Gradient clipping means just to put a hard cup C on the value of the gradient.

$$(\nabla E)_{new} = \min(\nabla E, C) \tag{2.11}$$

Note that the value of ∇E can only be positive as it is a vector and we always consider the direction of this vector as the optimal direction for an update. Cliff structures are most common in the cost functions for recurrent neural networks[2], that we will introduce in section section 2.3, because such models involve a multiplication of many factors, with one factor for each time step. Long temporal sequences thus incur an extreme amount of multiplication [18].

Another import problem in deep learning is the gradient loss. The gradient should reach zero value only when a minimum is reached, but recalling the chain rule in Eq. (2.9) we can see that there are other partial derivatives that can influence the gradient value. Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable. In particular we recall that activation function, such as sigmoid, are highly non linear, this can lead to a nullification of the gradient that jeopardises the training algorithm [18]. For this reason LearkyReLU Table 2.2 was developed, as in this particular function the gradient is never zero. LeakyReLU didn't ruled out gradient loss from deep learning, as this problem is always persisting the deeper network we have.

2.2 Principal Deep Learning layers

In this section we will introduce the principal Deep Learning layers used to build any deep learning model or architecture.

2.2.1 Fully connected Layer

Fully connected layer is actually just a perceptron layer. It is defined as fully connected because if we draw the graphs architecture every layer's unit is connected to every input, as seen in Figure 2.2. Remember that sometimes in software writing fully connected layers don't include an activation layer as this has to be specified.

2.2.2 Convolutional Layer

The name convolutional neural network indicates a network that employs convolution. Convolution s is a linear operation combining two functions x and w. It is defined as follows:

$$s(t) = \int x(a)w(t-a)da = (x * w)(t) \tag{2.12}$$

In deep learning we do not have continuous functions as we work solely with matrices, so we use instead the discrete formulation:

$$s(t) = \sum_{a=-\inf}^{\inf} x(a)w(t-a)da = (x * w)(t) \tag{2.13}$$

Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers. They are used primarily in image task as they have an inductive bias for 2D structure. They work by selecting a narrow window in the input and applying three passage. The window that slides across the inputs, as in Figure 2.3 until all computation are complete. We can modify every passage of this procedure by changing the size of the window or the shifting steps. The three passage are:

- Convolutional computation layer
- Activation function layer
- Pooling layer

The first step compute the convolution between the input matrix X and the weight matrix W, as the window slides through the inputs W matrix is always the same. This leads with respect to a fully connected layer in a sizeable reduction of weight parameters.

After the convolution we have the activation layer that add non linear capabilities, these are the classical activation function seen in Table 2.2.

The last step consist in a pooling function that replaces the output with a statistic summary. For example, the max pooling [22], just return us the max values between elements in the window. There are other pooling function that can be used such as L^2 norm or average pooling.

The use of pooling can be viewed as adding an infinitely strong prior that the function the layer learns must be invariant to small translations. When this assumption is correct, it can greatly improve the statistical efficiency of the network [23].

2.2.3 Residual Layer

Residual layers where first used in "Res_net" architecture [24]. Before this paper architectures were limited in depth because of gradient loss problems, the authors had a ground breaking idea, instead of after every layer re build all value, we ask a layer only to build up from the previous input. In practice we ask to calculate the residual and update the input value. Recalling Eq. (2.7) in mathematical term we can write the difference as:

$$\bar{y} = f(\mathbf{W}\bar{x}) \rightarrow \bar{y} = f(\mathbf{W}\bar{x}) + \bar{x} \tag{2.14}$$

This reduces the gradient loss because the residual connection act as a sort of skip, as in figure Figure 2.4 connection to skip non linear activation function that hindered the gradient quality.

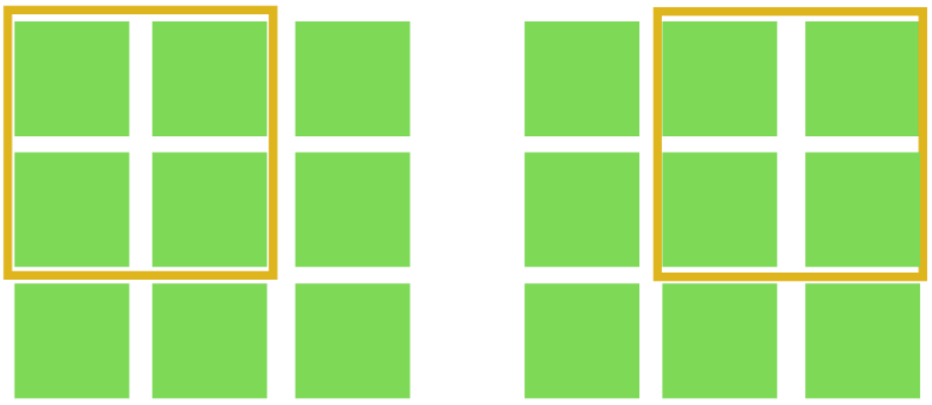

Figure 2.3: Sliding of the convolutional layer's widow in a 2D structure

2.2.4 Batch Normalization Layer

Batch Normalization [25] is a method of adaptive reparametrization, for almost any deep network, to increase model's depth. The goal is to standardize the output of a neuron. Previous approaches used penalties in the cost function to encourage neurons to have normalized activation statistics or renormalized neuron statistics after each gradient descent step [18]. This resulted in imperfect normalization or in a significant wasted time, as the learning algorithm repeatedly proposed changing the mean and variance. Batch normalization instead reparametrizes the model to make some neurons always be standardized by definition. For this Batch normalization has made models significantly easier to train. Given a batch **H** we normalize it as follows:

$$\mathbf{H}' = \frac{\mathbf{H} - \bar{\mu}}{\bar{\sigma}} \tag{2.15}$$

where $\bar{\mu}$ and $\bar{\sigma}$ are respectively the mean and standard deviation vectors of the layer. The idea here is to broadcast $\bar{\mu}$ and $\bar{\sigma}$ to every row of the matrix **H**. We define $\bar{\mu}$ and $\bar{\sigma}$ as:

$$\begin{aligned} \bar{\mu} &= \frac{1}{m}\sum_{i}^{m}\mathbf{H}_i \\ \bar{\sigma} &= \sqrt{\sigma + \frac{1}{m}\sum_{i}^{m}(\mathbf{H} - \bar{\mu})_i^2} \end{aligned} \tag{2.16}$$

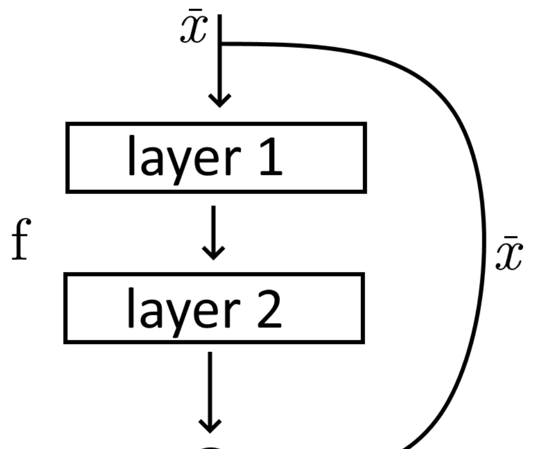

Figure 2.4: Residual layer [24]

2.3 Recurrent Neural Networks

Recurrent Neural Networks is a class of Neural Networks that process sequential data. The key idea behind is to have a dense representation of the input and update it throughout every element of the input vector. This is achieved with an hidden state is propagated in time. The particularity of such network is that weights of each layer are shared between time steps so the model is very light and it can have multiple inputs and outputs, for this we say it's particularly suited for sequenced data, in fact they were widely used in application such as Natural Language Processing and Time Series Forecasting. We refer to RNNs [2] as operating on a sequence, a time series, $\bar{x} = x_1, x_2, \ldots, x_\tau$ of length τ.

As RNNs are really flexible we provide in Figure 2.5 the computational graphs associated with this architecture.

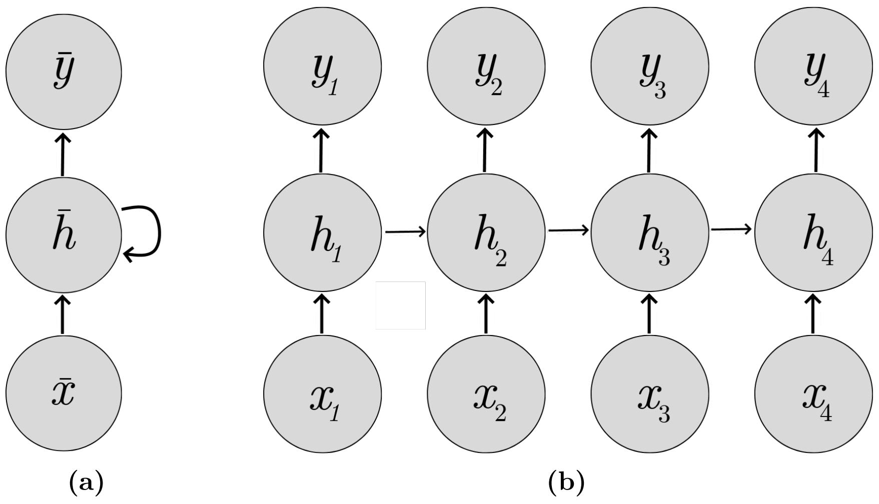

Figure 2.5: Recurrent neural network computational graphs, folded (a) and unfolded (b)

We can write the hidden representation as:

$$\bar{h}_t = f(\bar{h}_{t-1}, \bar{x}_t; \mathbf{W}) \tag{2.17}$$

Updating the hidden state h according to the Eq. above we see that it now contains information about the whole past sequence. This is an extreme simplification of RNNs, according to the first formulation of this type of architecture the following activation functions were used

$$\begin{aligned} \bar{h}_t &= \tanh \bar{\theta} + \mathbf{W}\bar{h}_{t-1} + \mathbf{U}\bar{x}_t + \bar{\theta} \\ \bar{y}_t &= \mathbf{V}\bar{h}_t + \bar{c} \end{aligned} \tag{2.18}$$

where $\bar{\theta}$ and $\bar{c}$ are the bias vectors and $\mathbf{W}, \mathbf{U}$ and $\mathbf{V}$ are weight matrices respectively, for input-to-hidden, hidden-to-output and hidden-to-hidden connections. For a graphical reference see fig Figure 2.5.

This particular formulation of RNNs is now called Vanilla RNN as to indicate it's original state, this is because sometimes "RNN" is used to indicate both "Vanilla RNN" or "RNNs". In this work we tried to differentiate between the two as much as possible.

RNNs provide some useful benefits, in fact regardless of the input length the learned model has always the same input size and it really light weight as it learns a single function f that operates on all time steps and all sequence lengths. This allows the model to work with dataset of different input length, for example with NLP datasets from two natural languages for translation tasks.

2.3.1 Vanilla RNN Problems

Vanilla RNN is the first architecture to use this shift in paradigm but depth of this fist architecture suffered form gradient loss as, according to (2.18) the gradient at each step passes through a hyperbolic tangent activation function. tanh suffers heavily from gradient loss as the further the input is from 0 the smaller the gradient is. In particular if the output layer is only in the last timestep the gradient has to travel all the way to the first input passing throw each tanh that adds the possibility of a gradient deterioration. Second for the recurrent nature of the architecture as it expand in horizontally, and not vertically, we cannot make use of the power of parallel computing that has yields so mush fruits in Deep Learning [26]. In practice, if there are multiple outputs gradient flow is stopped to retain some parallelization capabilities and reduce clock time for training models, but this aggravate the long term dependencies problem we are about to show.

As time series have generally long term dependencies, RNNs have particularly challenges in dealing with them. For if we imagine a simple RNN without the activation function or bias $\bar{h}_t$ state will be:

$$\bar{h}_t = \mathbf{W}^t \bar{x} = \mathbf{Q}\bar{\lambda}^t\mathbf{Q}^T\bar{x} \tag{2.19}$$

where $\mathbf{Q}$ and λ are respectively the orthonormal eigenvectors and eigenvalue. So for values $\lambda_i < 1$ the corresponding values of x_i will be lost. Furthermore, whenever the model is able to learn long-term dependencies, the gradient of these is exponentially smaller with respect to those of short term interaction. This means that is really hard and time consuming to learn long term interaction with RNN yet proposed. A solution has been developed in gated RNNs, such as LSTM [27].

2.3.2 LSTM

LSTM stands for Long Short Term Memory and were first introduced by Hochreiter and Schmidhuber in 1997 [27]. This architecture introduced the concept of a pass through for the gradient to back propagate freely along the model. For this reason has found many successful application in speach recognition [28], machine translation [29] and business analytic [30] just to cite a few areas. It is a similar concept to the residual layers introduced in section subsection 2.2.3, to let the gradient have a direct and linear path to all the inputs and resolve the vanishing gradient problem. LSTM have a fairly more complicated cell structure depicted in Figure 2.6 that is composed of various gating function:

$$\begin{aligned}
\bar{f}_t &= \sigma(\mathbf{U}_f\bar{x}_t + \mathbf{W}_f\bar{h}_{t-1} + \bar{\theta}_f)\\
\bar{i}_t &= \sigma(\mathbf{U}_i\bar{x}_t + \mathbf{W}_i\bar{h}_{t-1} + \bar{\theta}_i)\\
\bar{o}_t &= \sigma(\mathbf{U}_o\bar{x}_t + \mathbf{W}_o\bar{h}_{t-1} + \bar{\theta}_o)\\
\bar{c}_t &= \tanh \mathbf{U}_c\bar{x}_t + \mathbf{W}_c\bar{h}_{t-1} + \bar{\theta}_c\\
\bar{s}_t &= \bar{f}_t \circ \bar{s}_{t-1} + \bar{i}_t \circ \bar{c}_t\\
\bar{h}_t &= \bar{o}_t \circ \tanh s_t
\end{aligned} \tag{2.20}$$

where $\mathbf{U} \in \mathbb{R}^{h\times d}; \mathbf{W} \in \mathbb{R}^{h\times h}; \bar{\theta} \in \mathbb{R}^h$ are respectively input weights matrix, recurrent weights matrix and bias vectors and the superscripts d and h refer to the number of input features and number of hidden units.

Each of the function in 2.20 has a particulate purpose: $\bar{f}_t \in (0{,}1)^h$ is the forget gate, it regulates the amount of information to retain from the previous state $\bar{h}_{t-1} \in (-1{,}1)^h$; $\bar{i}_t \in (0{,}1)^h$ and $barc_t \in (-1{,}1)^h$ form the input gate that is responsible of extracting the information from the input vector $\bar{x}_t \in \mathbb{R}^d$ and previous hidden state vector $\bar{h}_{t-1}$
The cell state vector $\bar{s}_t \in \mathbb{R}^h$ represent the a dense representation of the LSTM cell,

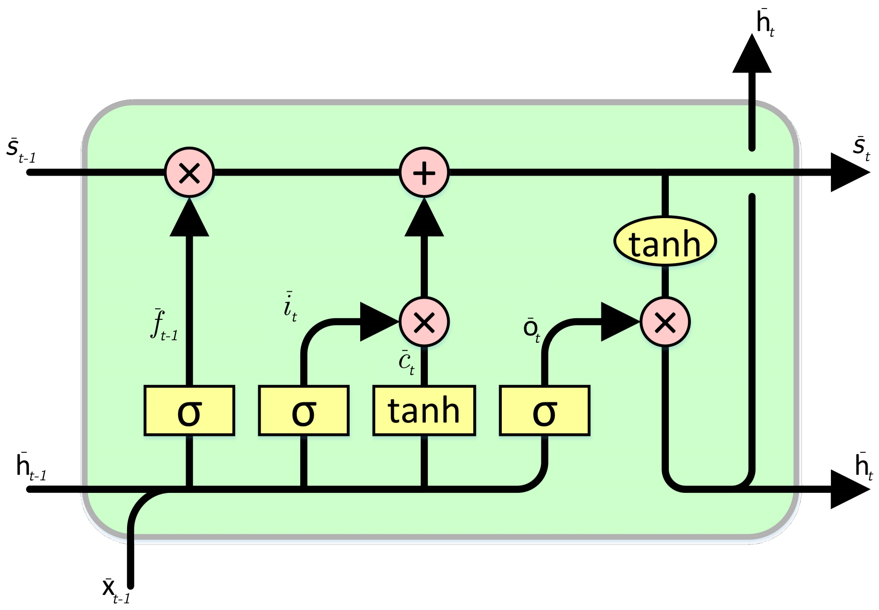

Figure 2.6: LSTM cell structure

it gets information both from the previous cell state and from the input, through respectively the forget gate and the input gate. $\bar{s}_t$ represent the great improvement of this architecture as the gradient can backpropagate between cell states without passing through non linear functions that could degrade it. At last we have the output gate $\bar{o}_t \in (0,1)^h$ that regulates the amount of information to disclosure to the hidden state.

This brief introduction to RNNs doesn't touch upon many aspect of this fascinating architecture. We ha seen only mono directional RNNs, but also bi directional variation have been developed. Also we have seen only the depth fixed at 1 while we can increase that number to add more hidden representation.

2.3.3 Time series forecasting with RNNs

When the recurrent network is trained to perform time series forecasting, the network typically learns to use the hidden state $\bar{h}$ as a sort of lossy summary of the task-relevant aspects of the input sequence [26]. So in practice, obviously, it tries learns the time series patterns we talked in previous chapter, section 1.1. Nowadays in Deep Learning for time series forecasting we have seen the rise of hybrid models, that combine statistical models seen in section 1.2 and deep learning models. This because it has been seen that for low data regime classical models perform better [31]. Hybrid models allow domain experts to inform neural network training using prior information. An example of this is the Exponential Smoothing RNN [32] that uses exponential smoothing [33],a classical decomposition method, to capture non-stationary trends and learns additional effects with the RNN [2].

More reference for state of the art RNNs applied to Time series forecasting can be found in "Time Series Forecasting With Deep Learning: A Survey" [31].

2.4 Transformer

Transformer architecture [1], shown in Figure 2.7, was introduced by Vaswani, it's an autoregressive architecture originally made for Natural Languages Processing. This paper sparked a revolution regarding most of Machine Translation but even other ares of Deep Learning research [34, 35].
Variation of the transformer are now used by the Big Four for most of their speech recognition services. And this type of architecture are being applied to other areas with various level of success.

2.4.1 Towards Transformer SeqtoSeq

Before introducing the transformer we need to talk about the SeqtoSeq[36] architecture. SeqtoSeq, stands for Sequence to sequence [36], and is a method that uses two multilayered LSTM, an encoder and a decoder (see Figure 2.8). The first maps the input sequence to a hidden representation of a fixed dimensionality, and then the second decode the hidden representation into an output sequence. The input and output structure is identical to the Transformer. We present first the machine translation setting as it's the original development task for both model. In machine translation the input vector is just a sequence of words (a sentence) that are fed to the encoder to develop the hidden representation we have discussed before. The decoder instead as we can see from Figure 2.8 has two inputs, the hidden representation and a start sentence token. In this case the decoder works in a dynamic manner, the start token is fed into the decoder that generates a word as the output. The first output word in then fed back into the decoder to iterate the process until an end sentence token is generated.
This can be easily generalize in the case of time series forecasting where as input

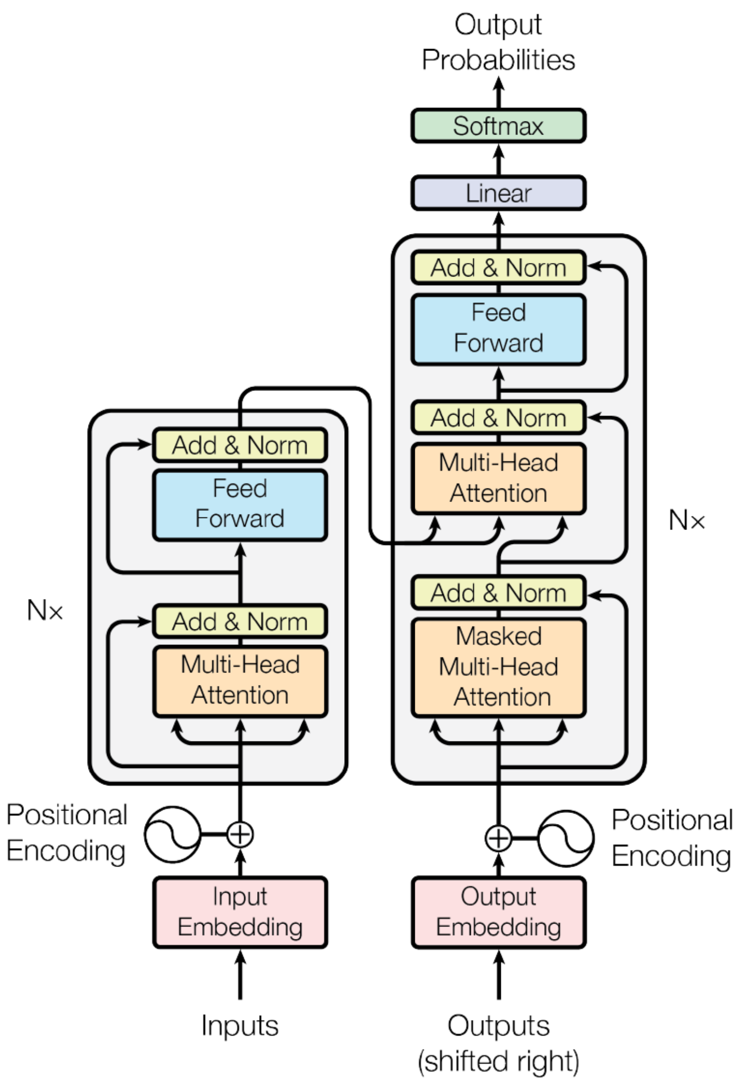

Figure 2.7: Transformer architecture[1]

of the encoder we have the time series and as output/input of the decoder the forecasting.

The key problem here is the all the information in condensed into the hidden representation $\bar{h}$, this is clearly a bottleneck of the SeqtoSeq model.

One idea to upgrade this model was to use also an attention mechanism [38], that we will explain in details in the following section subsection 2.4.3, to get around the bottleneck. The core idea is to let the decoder have a direct access at

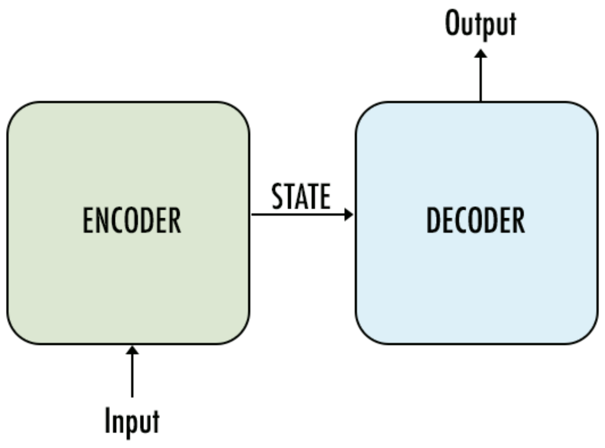

Figure 2.8: Encoder decoder architecture[37].

each input word of the encoder, so that at each time step the decoder can focus on a particular part of input sequence.
Attention improved significantly the performance and it helped with the gradient loss problems and the bottleneck. It also added more interpretability to the model, but it lacked still the scalability as RNN architecture are sequential. Ultimately this problem lead to the development of the Transformer[1] that uses a purely feed forward architecture.

2.4.2 Embedding

Before talking about the nuances of this architecture we have to introduce some key components. In machine learning embedding means to codify a data to a vector representation of different dimension. It is used for various purposes, for example as neural network cannot be fed with raw sequences of character we can use an embedding to encode the information into a numeric vector and proceed with the deep learning analysis.

In the Transformer architecture multiple embedding are used. Word2Vec [39] is used to transform words into vectors, then a second embedding, called positional embedding is used to give the model a spacial information. As we will see in the following section the Transformer's architecture is insensible to any operation of permutation on input. So to retain the sequential information of a sentence positional embedding has been developed.

The first positional embedding developed by the authors of Transformer [1] was fixed, meaning it doesn't change during training and is defined as following:

$$p_t^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k \cdot t), & \text{if } i = 2k \\ \cos(\omega_k \cdot t), & \text{if } i = 2k+1 \end{cases} \tag{2.21}$$

where:

$$\omega_k = \frac{1}{10000^{2k/d}} \tag{2.22}$$

Index t represents the position in the sequence of token word, while i goes from 1 to d, the dimension of the embedding. If i is even we will use the sine function, while if it is odd we will use the cosine. ω is called the frequency. The authors used this particular combination because relative positioning can be express simply by a linear function.

$$\bar{p}_{t+\phi} = \mathbf{M}_{\phi}\bar{p}_t \tag{2.23}$$

We can make a simple example with dimension 2, it can be proven that:

$$\mathbf{M}_{\phi,q} = \begin{bmatrix} \cos(\omega_k \cdot \phi) & \sin(\omega_k \cdot \phi) \\ -\sin(\omega_k \cdot \phi) & \cos(\omega_k \cdot \phi) \end{bmatrix} \tag{2.24}$$

So Equation 2.23 becomes:

$$\begin{bmatrix} \sin(\omega_k \cdot (t+\phi)) \\ \cos(\omega_k \cdot (t+\phi)) \end{bmatrix} = \begin{bmatrix} \cos(\omega_k \cdot \phi) & \sin(\omega_k \cdot \phi) \\ -\sin(\omega_k \cdot \phi) & \cos(\omega_k \cdot \phi) \end{bmatrix} \begin{bmatrix} \sin(\omega_k \cdot t) \\ \cos(\omega_k \cdot t) \end{bmatrix} \tag{2.25}$$

Learned positional embedding instead are simply a fully connected layer, so they are learned by the model during training, exploiting the modern data driven paradigm. Both fixed and learned positional embedding are simply added to the output of the input embedding as seen in Figure 2.7.

2.4.3 Attention

The attention mechanism has a key role in the Transformer architecture. It is a mechanism that aims to enhance some characteristic of the input as oppose

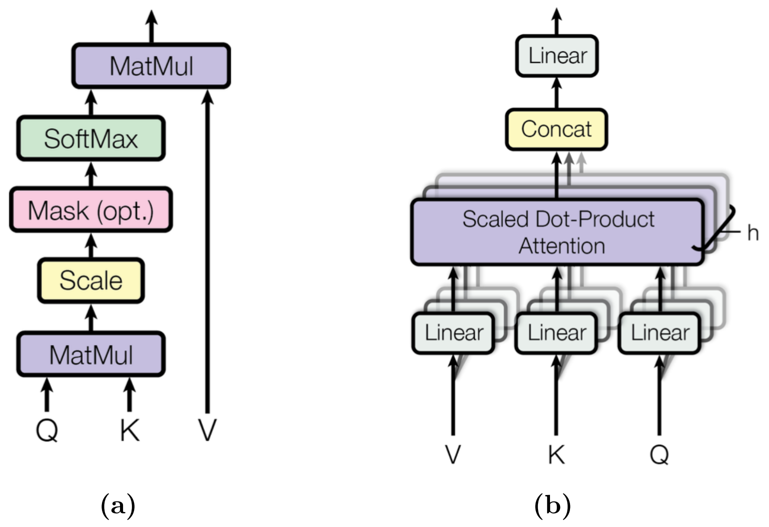

Figure 2.9: Attention block of Transformer, scaled dot product attention (a) and multi head attention (b)[1]

to other. We have three input matrices $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$, respectively named queries, keys and values. The idea is simple, for each query vector we search for similarity in the key matrix. For each key vector is associated a single value vector. So we weight each value with a function of the similarly between our query and the corresponding key of the value. In formulas,

$$\text{Att} = f(\mathbf{Q}, \mathbf{K})V \ \ , \qquad f(Q, K) = softmax(\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{d}})\mathbf{V} \tag{2.26}$$

Where $\mathbf{V}, \mathbf{K}, \mathbf{Q} \in \mathbb{R}^{L \times d}$, d is the embedding dimension and L the input sequence length. The use softmax function to weight the values has been proposed by the authors, but it has led to some substantial problems of computational complexity we will address in Chapter section 3.2. The $\sqrt{d}$ term is there just for regularization purposes and doesn't change the outcome of the weighted sum.
We can re write the attention in a more clever way as

$$\text{Att}_{\leftrightarrow}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{D}^{-1}\mathbf{A}\mathbf{V} \tag{2.27}$$

where we define

$$\begin{aligned} \mathbf{A} &= \exp(\mathbf{Q}\mathbf{K}^{\mathrm{T}}/\sqrt{d}), \\ \mathbf{D} &= \operatorname{diag}(\mathbf{A}\mathbf{1}_L) \end{aligned} \tag{2.28}$$

where $\mathbf{1}_L$ is just a unitary vectory of dimention L. The above attention is also called bidirectional as information flows in both direction with respect to the input sequence. As opposed to bidirectional attention we can think of a single directional attention (or casual attention) where information comes only from past tokens. Imagine input vector at time t can access only previous keys. This is used during inference, in the decoder, to deny the possibility of the transformer architecture to access future values [26].
To reach this goal we just multiply a triangular matrix tril of 0,1 to block future entry scores. Thus, we can write the attention matrix as

$$\mathrm{Att}_{\rightarrow}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \widetilde{\mathbf{D}}^{-1}\widetilde{\mathbf{A}}\mathbf{V} \tag{2.29}$$

where we define

$$\begin{aligned} \widetilde{\mathbf{A}} &= \mathrm{tril}(\mathbf{A}), \\ \widetilde{\mathbf{D}} &= \mathrm{diag}(\widetilde{\mathbf{A}}\mathbf{1}_L) \end{aligned} \tag{2.30}$$

2.4.4 Encoder

As we can see form the Figure 2.7 this is the full architecture of the transformer. It presents an encoder-decoder structure just like SeqtoSeq [36].
The encoder takes the input matrix as a sum of the various embedding layers. The input than goes to the multi-head attention layer, seen in detail in Figure 2.9. This layer is just a multiple attention computed in parallel, the authors decided to split the computation of the attention matrix reducing the hidden dimensions h of each query, keys and values. This split permits at each head to focus it's attention into various feature on the input values.

Let $h = e/d$, with e and d indicating respectvelly the embedding dimension and the h hidden dimension of the attention, and $X \in \mathbb{R}^{L \times e}$. For each head i, with $\{i \in \mathbb{N}, i < h\}$, the attention $\mathrm{Att}^i_{\leftrightarrow}$ is computed as:

$$\mathrm{Att}^i_{\leftrightarrow} = \mathrm{Att}^i_{\leftrightarrow}(\mathbf{W}^Q_i\mathbf{X}, \mathbf{W}^K_i\mathbf{X}, \mathbf{W}^V_i\mathbf{X}) \tag{2.31}$$

where $\mathbf{W}^V_i, \mathbf{W}^K_i, \mathbf{W}^Q_i \in \mathbb{R}^{e \times d}$ are values, keys and queries weight matrices, representing dense layers. The output of all heads is concatenated as

$$\mathrm{Out} = \mathbf{W}^C\mathrm{Concat}(\mathrm{Att}^1_{\leftrightarrow}, \ldots, \mathrm{Att}^h_{\leftrightarrow}) \tag{2.32}$$

where $\mathbf{W}^C$ represents the weight matrix of another dense layer. We can see also from the image Figure 2.7, and as reported in the above equation, that values, keys and queries are formed as a linear combination of the input matrix. For this reason this attention, used in Transformer [1], it is also called self attention as the score attention matrix originate from the input matrix and just weights the input it self.

Following Figure 2.7 after the multi-head attention layer there is a Feed Forward layer. This is a simple block containing in series a fully connected layer, a ReLU activation function, and another fully connected layer. The hidden dimension h in this block is expanded 4 times before the ReLU and then compress again to retain the original dimensions. This is a choice of the authors, and other researcher [40] have selected a opposite approach.

After each of this two blocks the output matrix goes to a batch normalization layer and a residual layer.
This conclude the encoder architecture that can be repeated multiple times in series. The output of the encoder is then pass in the decoder to further use in the cross attention.

2.4.5 Decoder

The decoder, as we can see from the Figure 2.7, share some of the blocks architecture with the encoder. We can see there are two separate attention blocks, both with some minor differences with respect to to the encoder one. The first it's called causal attention as it uses a triangular mask to enforce causality on the decoder input as explained in the previous section in Eq. (2.30). Also in this case, it is called self attention and the matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are functions of the decoder input. Instead the second attention block is no longer a self attention, but it exploits the hidden representation of the encoder, i.e. the output of the encoder, to compute keys and values. Queries are still a function of the decoder input because the idea is similar to SeqtoSeq with attention [36, 38] that gives the decoder direct access to the encoder input. For this reason the latter attention is also called cross attention.
Just like with the encoder, after each block of attention or feed forward NN there are a batch normalization layer and a residual layer. Also the decoder structure can be repeated multiple times and the output goes directly to resolve the selected task, prior some shift in dimensional space or activation function specific for task.

2.4.6 Training and Inference

The training procedure uses an optimization algorithm, as seen in section subsection 2.1.3, with the loss function computed with the target sequence and the decoder output. For the inference procedure, recalling that even in training we need to do inference to compute decoder's output, we can have two different approaches: Dynamic inference or Teacher Forcing.
Dynamic inference introduced in section 1.3 means to output a single data point at iteration and then insert the new point as the decoder input to repeat the process until the end. If we think about it, if there is an error at timestep t, with dynamic inference we will propagate this error for successive timesteps. This is particularly present in a naive approach that minimizes the loss at each point directly, some improvement can be achieved using more sophisticated strategies such as Beam search [41].
While in the real setting, with the model deployed, inference can be only carried out with a dynamic fashion, but during training we can use another approach called Teacher forcing.
Teacher forcing forces the decoder input to be the ground truth of target sequence. This adds come major benefits and drawbacks. First we resolve the problem of propagating the error into successive timesteps as any error will be confined to the output. But this adds a bias to our network as deployment condition are different to training. In practice for transformers teacher forcing is heavily use in Neural Machine Translation as it has the added benefit to compute the output in a single sweep, as no dynamic process is needed, and this reduces substantially training times.

2.4.7 Time series forecasting with Transformers

Originally the transformer was used only for Neural Machine Translation [1] but other variation have been proposed to extend it's used also in time series forecasting [35, 5, 42]. Transformer doesn't scale well with Multivariate time series so we need to modify the architecture, we will go in depth into this problem in next chapter 3, for now take this in mind.
Obviously we need also to replace the encoding, NLP taks use Word2Vec [39] encoding that is out of context with Time series forecasting. We typically substitute it with a more appropriate Time2Vec [43]. Time2Vec is an embedding of size $k+1$ that take as input a scalar time value t, it has been designed by the authors to be time rescaling invariant, means it can work without differences with timeseries which time step unit is seconds and ones which have hours. Time2Vec is defined

as:

$$v_t^{(i)} := \begin{cases} \omega_i t + \theta_i \text{ if } i = 0 \\ \mathcal{F}(\omega_i t + \theta_i) \text{ if } 1 \leq i \leq k \end{cases} \tag{2.33}$$

It is somewhat similar to positional encoding defined in Eq. (2.21) but in this case function $\mathcal{F}$ is a periodic activation function and ω_i and θ_i are learnable patterns. The authors of Time2Vec [43] proved that this embedding is able to learn any periodic pattern with different periodic functions such as sin, $\mod$, triangle, with sin outperform the other two.
Another note is that the hyperparameter k defines the number of periodic pattern to be learned, and the first component $v_t^{(0)}$ encodes the liner scaling of time.

As we have seen in section 1.3, the forecasting can be carried out with a single sweep fashion (without teacher forcing) or in dynamic inference, the literature is exploring both option. With a single sweep Transformer architecture can be modified to include only the encoder as the output is always of fixed length. While in a dynamic environment a decoder section is always needed. In the next chapter we will go in depth Informer architecture[5] as it is the backbone of this thesis work.

Chapter 3

Problems and solution in Transformer literature

3.1 Complexity and architecture evaluation

In computer science, the computational complexity or simply complexity of an algorithm is the amount of resources required to run it. In particular time complexity and memory space complexity are the focus of these type of analysis. We will typically measure the computational efficiency algorithm, or time complexity, as the number of a basic operations it performs as a function of its input length L [44]. The computational complexity of training a deep neural network model depends on multiple factors. If we focus only on the training task, the more demanding phase with respect to deployment, is composed of a feed forward phase, to compute the output, and a backpropagation phase, to update weights matrices [18].
It can be quite challenging to estimate the complexity, but the big $\mathcal{O}$ notation give us a hand. Big $\mathcal{O}$ notation ($\mathcal{O}(L)$) is a mathematical notation that describes the behavior of a function as it's inputs grow larger and larger.

Given two function $f, g : \mathbb{N} \to \mathbb{N}$, we say that $f = \mathcal{O}(g)$ or more explicit $f(L) = \mathcal{O}(g(L))$ if there exist a constant c such that $f(L) \leq g(L)$ for $L \to \infty$.

To train a model for an epoch means to train it for a single training datasets pass. In deep learning we focus on single epoch complexity, this is because we do not want to evaluate the optimization algorithm but the architecture.

Self attention	$\mathcal{O}(L^2 \cdot d)$
Recurrent NN	$\mathcal{O}(L \cdot d^2)$
Convolutional NN	$\mathcal{O}(L \cdot d^2 \cdot kernel\_width)$

Table 3.1: Computational complexity of Attention, RNN, Convolutional NN

But complexity is just a theoretical measure, so in real world settings models are also evaluated through their wall clock time and CPU (or GPU) time. Wall clock time time measures how much time has passed, as if you were looking at the clock on your wall, while CPU time is how many seconds the CPU was busy [45].

After this brief introduction on architecture evaluation we can now focus our attention on the Transformer [1].

3.2 Problems of attention matrix

As we have seen in the definition of the attention matrix in Eq. (2.28), $\mathbf{A} \in \mathrm{R}^{L \times L}$ where L is the input length. So to compute the full matrix we have a complexity that scales quadratically with respect to L, i.e.

$$\mathcal{O}(L^2 \cdot d) \tag{3.1}$$

We can see the difference with respect to other architectures we have introduced in the previous chapter 2 in Table 3.1.

This means that the Transformer, that has an attention mechanism, becomes unfeasible to train for high values of L. This may not be a problem for some task of NLP, such as machine translation, but it becomes one for other task, like document sentiment analysis or any task of computer vision. In fact if we think about images, L is the number of pixels, that is notoriously high. If we think about ImageNet [46], the most famous image database, its images are usually cropped at 256x256 pixels. Thus, if we tried to feed the full image to a Transformer, L would be $L = 256 \times 256 = 65.536$.

Model/Paper	Complexity	Class
Image Transformer[34]	$\mathcal{O}(L \cdot m)$	FP
Set Transformer[48]	$\mathcal{O}(L \cdot k)$	M
Sparse Transformer[49]	$\mathcal{O}(L\sqrt{L})$	FP
Reformer[50]	$\mathcal{O}(L\log(L))$	LP
Routing Transformer[51]	$\mathcal{O}((L\log(L))$	LP
Performer[3]	$\mathcal{O}(L)$	FP
Linear Transformers[52]	$\mathcal{O}(L)$	FP
Big Bird[4]	$\mathcal{O}(L)$	FP+M

Table 3.2: Summary of Efficient Transformer Models presented in chronological order of their first public disclosure. Class abbreviations include: FP = Fixed Patterns or Combinations of Fixed Patterns, M = Memory, LP = Learnable Pattern and KR = Kernel. Where m and k are particular parameters of the relative architecture. This is a subset the full table found in "Efficient Transformers: A Survey" [47]

3.3 Efficient Transformers

Various attempts have been made in literature to improve the original architecture of Transformers, we have decided to classify this attempts into three broad categories:

- Attention approximation
- Architectural changes
- Inference optimization

3.3.1 Attention approximation

This first category consist in those innovation whose aim is to approximate the full attention computation reducing the consequence the complexity of the computation.
We have multiple types of approximation and architecture that are summed up in "Efficient Transformers: A Survey"[47]. In Table 3.2 we present a subset of the results authors' taxonomy study.

Fixed Patterns Are earliest type of modifications to self-attention. It simply sparsifies the attention matrix by limiting the field of view to fixed, predefined patterns such as local windows and block patterns of fixed strides.
"Sparse transformer" [49] and "Image Transformer" [34] are example of this category. Also "Big Bird" [4] architecture uses some feature that can be collocated in this group, we will see in detail Big Bird architecture in section 3.5.

Memory Another prominent method is to leverage a side memory module that can access multiple tokens at once. A common form is global memory which is able to access the entire sequence. The global tokens act as a form of memory that learns to gather from input sequence tokens.
This was first introduced in "Set Transformers" [48] as the inducing points method. These parameters are often interpreted as “memory” and are used as a form of temporary context for future processing. Big Bird [4] used also some memory feature.

Learnable Patterns The natural evolution to fixed, pre-determined pattern are learnable ones. Models using learnable patterns aim to learn the access pattern in a data driven fashion. A key characteristic of learning patterns is to determine a notion of token relevance and then assign tokens to buckets or clusters. The key idea of learnable patterns is still to exploit fixed patterns. However, this class of methods learn to sort/cluster the input tokens enabling a more optimal global view of the sequence while maintaining the efficiency benefits of fixed patterns approaches.
Some examples are "Routing Transformer" [51] and "Reformer" [50].

Kernels A recently popular method to improve the efficiency of Transformers is to view the attention mechanism through the kernel trick. The usage of kernels enable clever mathematical re-writing of the self-attention mechanism to avoid explicitly computing the $L \times L$ matrix. The idea is the same as we have seen in subsection 2.1.1 for the perceptron. In that case we did not want to compute the mapping of each entry to a higher dimensional space so we directly computed the dot product between entries skipping the mapping. In this case we what to skip the computation of attention matrix A as it require a great deal of resources.
Some examples are "Linear Transformers" [52] and "Performer" [3], we will see in detail the latter in section 3.6.

3.3.2 Architectural changes

For architectural changes we mean all those changes that modify the architecture of the original Transformer [1] without touching the attention matrix.

Parameters reduction In this case, we considered those researches whose objective is to reduce as much as possible the dimension of the network by reducing parameters of the model. It can be considered a vague and not so well defined category but as most efficient improvements reduce parameters, but in this case we are only talking about parameters outside the attention mechanism.
The most representative example is "DeLighT" architecture [40], that compresses the dense representation thanks to a deep and light-weight transformation on inputs variables.

Pooling Consists in adding a convolutional neural network layer to reduce significantly the length of the input. It is called pooing referencing the polling mechanism of CNNs that reduces the input length, it can be considered a type of fixed pattern as it applies always the same transformation to the input.
Some notable examples are "Scalable Vision Transformer" [55] and "Funnel Transformer" [56].

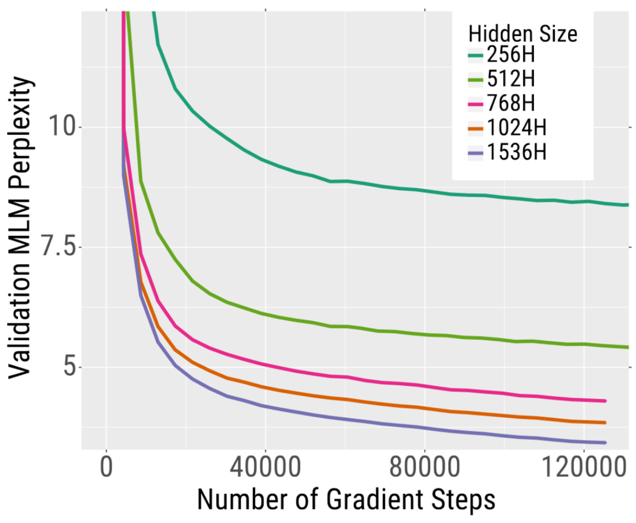

Figure 3.1: Performance of RoBERTa[53] Transformer with increasing depth plotted against gradient steps. Images taken from "Train Large, Then Compress" [54]

3.3.3 Inference optimization

Inference optimization are some technique that reduce the parameters and increase the speed in inference without reducing too much the performance. It is studied in detail in two separate papers "Train Large, Then Compress" [54] and "FastFormers" [57] independently written.
The authors of "Train Large, Then Compress" [54] proves empirically that wider and deeper Transformer models are more sample efficient than small models, meaning that they reach the same level of performance using fewer gradient steps (Figure 3.1). This increase in convergence outpaces the additional computational overhead from increasing model size at least in the NLP setting.

But having a larger model for inference is just not worth, naturally the authors began thinking about compressing the larger model to a smaller one and perform inference with the latter. Three are copression methods explored by the authors [54], [57]:

Quantization Low precision inference or quantization stores the model weights in low precision format, this means faster inference time with specific hardware that support low precision operation. We recall that modern framework stores weights in 32bit presition. The performance hit is negligible if we use 8bit or more, in some use cases even 6bit are sufficient[54].

Pruning It means to set to zero some weight of the model to decrease inference time. The have been various researches for pruning with different procedure, some random [58] other more structured [57], [59] that focus their attention on specific heads in the multi head attention mechanism, (see subsection 2.4.3).

Knowledge Distillation it is a well known model compression technique where the large model is used as a teacher for a smaller student model to mimic its behaviour. If we use pre-trained models we can apply knowledge distillation before of after fine tuning the model for a specific dataset [57].

3.4 Informer

The goal of this thesis is to explore the optimization of efficient attention on a time series forecasting task. For this reason, after extensive research over all the possible categories of efficient attention candidates, we decided to proceed with "Performer" [3], "Big Bird" [4] and "ProbSparse" [5] attentions. As introduced in the previous section they are respectively a Kernel, Fixed patter and Learnable pattern optimization (see subsection 2.4.3).
We also needed to provide backbone architecture, of the Transformer family, that could perform time series forecasting. We settled upon the "Informer" [5] that has a similar structure to the original transformer.

3.4.1 ProbSparse

The authors of Informer performed a qualitative assessment on the learned attention patterns of the canonical self attention, assessing that only few dot-product pairs contribute to the attention, and others generate low scores.
Based on this consideration they proposed ProbSparse Attention. This can be considered a learnable pattern attention according to the previous taxonomy. It is based on a sparcity measurement that aims to find the dominant dot-product pairs to compute them regardless of the other pairs. The idea is that given a query vector $\bar{q}_i$ then we can define $p(\bar{k}_j|\bar{q}_i)$ as the probability of distribution over $\bar{k}_j$. If $p(\bar{k}_j|\bar{q}_i)$ is close to a uniform distribution $g(\bar{k}_j|\bar{q}_i)$ then we can assume that the query is not important. This is measured by:

$$M(\bar{q}_i, \mathbf{K}) = \ln\left(\sum_j e^{\frac{\bar{q}_i \bar{k}_i^{\mathrm{T}}}{\sqrt{d}}}\right) - \frac{1}{L}\sum_j \frac{\bar{q}_i \bar{k}_i^{\mathrm{T}}}{\sqrt{d}} \tag{3.2}$$

However, calculating $M(\bar{q}_i, \mathbf{K})$ for all the queries requires calculating each dot-product pairs, i.e., quadratically $\mathcal{O}(L^2)$ so the authors proposed an empirical approximation of M:

$$\hat{M}(\bar{q}_i, \mathbf{K}) = \max_j \left\{\frac{\bar{q}_i \bar{k}_i^{\mathrm{T}}}{\sqrt{d}}\right\} - \frac{1}{L}\sum_j \frac{\bar{q}_i \bar{k}_i^{\mathrm{T}}}{\sqrt{d}} \tag{3.3}$$

According to $\hat{M}$ the top u queries are selected with $u = c \cdot \ln L$, where c is an hyperparameter of the model. Moreover, to estimate $\hat{M}(\bar{q}_i, \mathbf{K})$, we only need $U = L\ln(L)$ so the total complexity drops to $\mathcal{O}(L\ln(L))$. With the use of multi-head attention as each attention can focus on different part of the input, so the top u queries changes for each sub matrix, this mitigates the information loss of the sparsity.

3.4.2 Encoder

The overall structure of the Informer is very similar to the original transformer, as we can see from Figure 3.3.

The encoder after an attention block instead of using a two dense layer, like the classic transformer, it applies a convolutional 1D layer to compress the representation. As a result instead of having an attention matrix $\mathbf{A}_0 \in \mathbb{R}^{L\times L}$ for the second block we would have a matrix $\mathbf{A}_0 \in \mathbb{R}^{(L/2)\times(L/2)}$. The authors provided a schematic for an encoder block of depth 4 in Figure 3.2.

3.4.3 Decoder

The proposed decoder predicts outputs by one forward procedure rather than the time consuming dynamic decoding procedure of the conventional encoder-decoder architecture. A detailed performance comparison is given in the Informer [5] paper. So the input matrix of the decoder is $\mathbf{X}_{de} = \{\mathbf{X}_T, \mathbf{X}_0\}$ where T indicates a time

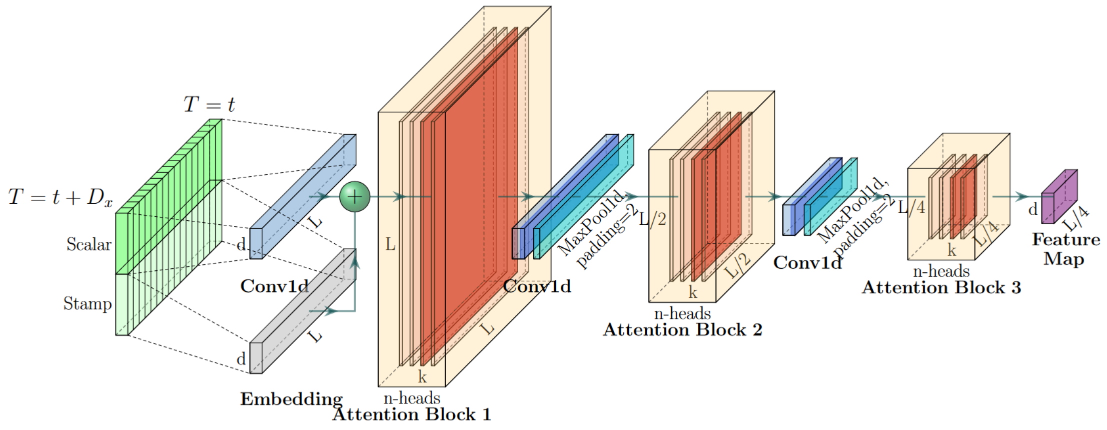

Figure 3.2: The single stack in Informer's encoder of depth 4 [5].

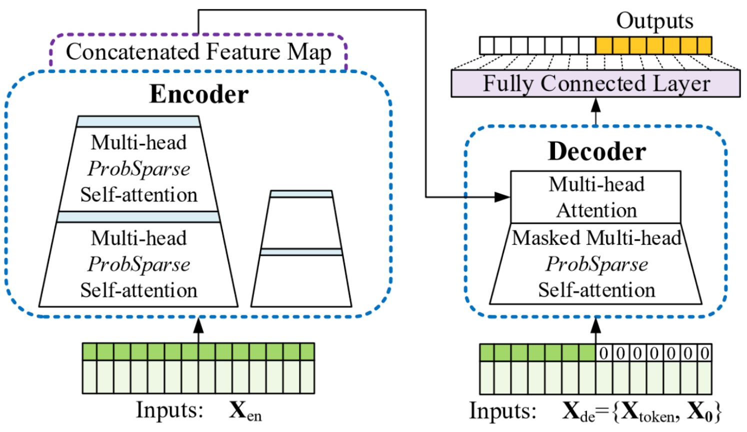

Figure 3.3: Informer architecture schema [5].

interval, while $\mathbf{X}_0$ are just the time stamp information without the values for the various feature. The overall architecture schema can be appreciated in Figure 3.3.

3.5 BigBird Attention

"BigBird: Transformers for Longer Sequences"[4] introduces some theoretical results that proves how sparcity can effectively approximate the full attention mechanism, then the author proceeds to develop a sparse attention that has proven to be the best of type fixed pattern incorporating some memory features, see Figure 3.4.

BigBird Attention is a fixed pattern sparse attention that has three patterns:

- Random attention: Each query block attends to r random key blocks.
- Window local attention: Every query block with index j attends to key block with index $j-(w-1)/2$ to $j+(w-1)/2$, including key block j.
- Global attention: that are specialized tokens, added or already existing that are used to capture the global context of the input.

For an efficent GPU/TPU harware implementation the authors first blockify the attention pattern, so pack sets of query and keys together and then define attention on these blocks. If we look at Figure 3.4 every single square is a block that can be composed by multiple keys and queries according to the block size

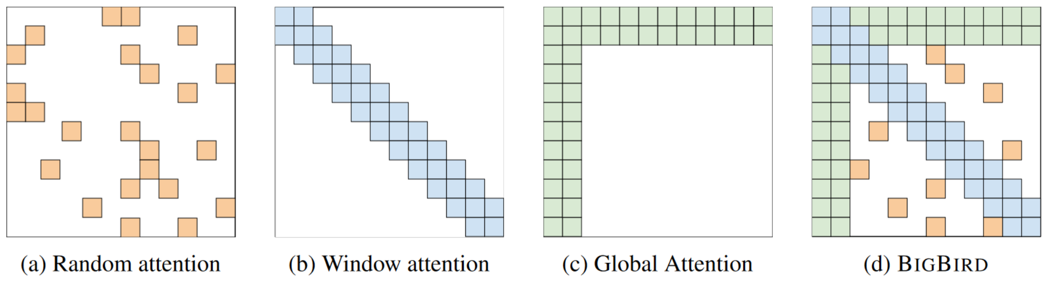

Figure 3.4: Building blocks of the attention mechanism used in BigBird. White color indicates absence of attention. (a) random attention with $r = 2$, (b) sliding window attention with $w = 3$ (c) global attention with $g = 2$. (d) the combined BigBird model. Image taken from "BigBird: Transformers for Longer Sequences"[4].

b. This procedure make possible to reshape the blocks to perform a single matrix multiplication for each attention component.
So the total computational cost of BigBird is $\mathcal{O}(Lkbd)$ where $k = (g + w + r)$, L is the input length, g the number of global tokens, w the dimension of the local window, r the random.

The paper also feature a theoretical proof of the approximation power of sparse attention.

3.6 Performer

Performer efficient attention mechanism, called FAVOR+ [3] (Fast Attention Via positive Orthogonal Random features), uses the kernel trick with positive orthogonal random mapping to decompose the attention multiplication and reduce drastically both time and space complexity.
We recall previous Attention definition seen in Equation 2.28 as:

$$\mathrm{Att}_{\leftrightarrow}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{D}^{-1}\mathbf{A}\mathbf{V} \tag{3.4}$$

where

$$\begin{aligned} \mathbf{A} &= \exp(\mathbf{Q}\mathbf{K}^{\mathrm{T}}/\sqrt{d}), \\ \mathbf{D} &= \mathrm{diag}(\mathbf{A}\mathbf{1}_L). \end{aligned} \tag{3.5}$$

We can write $\mathbf{A}$ as

$$\mathbf{A}(i, j) = \mathrm{K}(\bar{q}_i^{\mathrm{T}}, \bar{k}_j^{\mathrm{T}}) \tag{3.6}$$

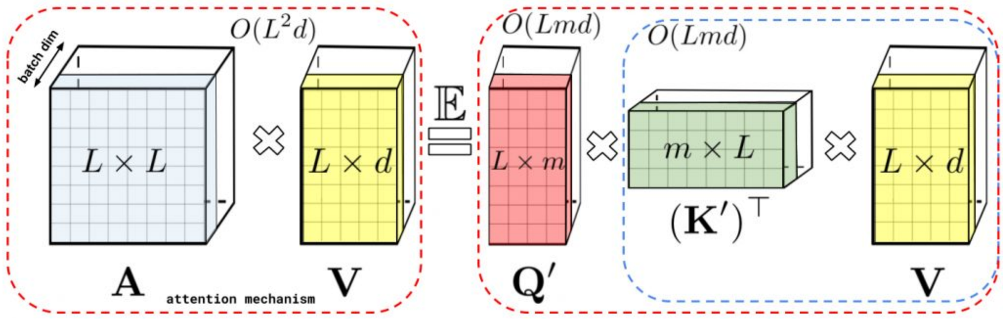

Figure 3.5: Approximation of the regular attention mechanism via (random) feature maps. Dashed-blocks indicate order of computation with corresponding time complexities attached [3].

with $\mathbf{A} \in \mathbb{R}^{L\times L}$, where K is the kernel function defined with random feature map $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^r$ with $r > 0$, as

$$\mathrm{K}(\bar{x}, \bar{y}) = \mathbb{E}[\phi(\bar{x})^{\mathrm{T}}\phi(\bar{y})] \tag{3.7}$$

We then can define $\mathbf{Q}'$, $\mathbf{K}' \in \mathbb{R}^{L\times r}$ the mapped matrix by ϕ where rows are respectively $\phi(\bar{q}_i^{\mathrm{T}})^{\mathrm{T}}$ and $\phi(\bar{k}_i^{\mathrm{T}})^{\mathrm{T}}$. We note that the expected value is computed over the random feature, we will see what this means in a bit.

Using the introduced keys and queries, we can rewrite the attention as

$$\widehat{\mathrm{Att}_{\leftrightarrow}}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \widehat{\mathbf{D}}^{-1}(\mathbf{Q}'((\mathbf{K}')^{\mathrm{T}}\mathbf{V})), \tag{3.8}$$

with $\widehat{\mathrm{Att}_{\leftrightarrow}}$ indicating the resulting attention approximation, where

$$\widehat{\mathbf{D}} = \mathrm{diag}(\mathbf{Q}'((\mathbf{K}')^{\mathrm{T}}\mathbf{1}_L)) \tag{3.9}$$

A visualization of this passage can be found in Figure 3.5. We can now understand how the FAVOR+ attention mechanism reduces the complexity from $\mathcal{O}(L^2d)$ to $\mathcal{O}(Lrd)$.

Going more into details, to understand what it actually means having a random feature mapping, first we need to write the softmax function SM as

$$\mathrm{SM}(\bar{x}, \bar{y}) \overset{\mathrm{def}}{=} \exp(\bar{x}^{\mathrm{T}}\bar{y}) = \exp\left(\frac{||\bar{x}||^2}{2}\right)K_{gauss}(\bar{x}, \bar{y})\exp\left(\frac{||\bar{y}||^2}{2}\right) \tag{3.10}$$

where K_{gauss} is the gaussian kernel. Note that we are omitting all the scaling constant to provide a simpler and easy to understand formulation.
The gaussian kernel can be approximated by a random feature map ϕ of type:

$$\phi(\bar{x}) = \frac{1}{\sqrt{m}}(\sin(\mathbf{W}^{\mathrm{T}}\bar{x}), \cos(\mathbf{W}^{\mathrm{T}}\bar{x})) \tag{3.11}$$

where matrix $W \in \mathbb{R}^{d\times m}$ is composed of m random vector w all identically distributed according to a normal distribution $\mathcal{D} = \mathcal{N}(0, \mathbf{I}_d)$. This is the results of a fascinating paper "Random Features for Large-Scale Kernel Machines" [60], but for our purposes it means that also the softmax function can be approximated using random mapping. Note that the expected value we found in Equation 3.7 it is computed over the vectors w.
The authors also introduced the constrain of orthogonality into random vectors w, this achieved by the "Gram-Schmidt orthogonalization procedure" [61] and has the scope of regularizing the kernel approximation.

The paper "Rethinking attention with Performers" [3] presents many possible approximation of the softmax attention derived by different mapping, in our case we will use the $\widehat{\mathrm{SM}}_m^+(\bar{x}, \bar{y})$ derived by mapping:

$$\phi(\bar{x}) = \frac{\exp(-\frac{|\bar{x}|^2}{2})}{\sqrt{m}}(\exp(\mathbf{W}^{\mathrm{T}}\bar{x})) \tag{3.12}$$

But how close is the approximation to the real value of the softmax attention? We can have a pretty good understanding of this computing the mean squared error between the two:

$$\mathrm{MSE}(\widehat{\mathrm{SM}}_m^+(\bar{x}, \bar{y})) = \frac{\exp(||\bar{x}+\bar{y}||^2)\mathrm{SM}(\bar{x}, \bar{y})^2(1-\exp(-||\bar{x}+\bar{y}||^2))}{2m} \tag{3.13}$$

We can denote that the means square error of $\widehat{\mathrm{SM}}_m^+$ is proportional to the softmax value squared, this means that with SM $\to 0$ also $\mathrm{MSE}(\widehat{\mathrm{SM}}_m^+) \to 0$. This important as softmax value has range between [0,1] and in particular in the attention mechanism where many pairs have a value close to 0 as we have seen in the discussion of ProbSparse attention in section 3.4.

Attention	Space Complexity	Time Complexity
Classical Attention	$\mathcal{O}(L^2 + Ld)$	$\mathcal{O}(L^2 d)$
ProbSparse Attention	$\mathcal{O}(L\ln(L) + Ld + \ln(L)d)$	$\mathcal{O}(Ld\ln(L))$
BigBird Attention	$\mathcal{O}(Lkb + Ld + Lkd)$	$\mathcal{O}(Lkbd)$
FAVOR+ Attention	$\mathcal{O}(Lr + Ld + rd)$	$\mathcal{O}(Lrd)$

Table 3.3: Complexity for every attention used in this thesis experiments

Model	1K	2K	3K	4K
Transformer	8.1	4.9	2.3	1.4
BigBird	7.4 (0.9x)	3.9 (0.8x)	2.7 (1.2x)	1.5 (1.1x)
Performer	9.5 (1.2x)	9.4 (1.9x)	8.7 (3.8x)	8.0 (5.7x)

Table 3.4: Steps per second performance in increasing Length input in "Long range Arena"[62]

Model	1K	2K	3K	4K
Transformer	0.85	2.65	5.51	9.48
BigBird	0.77 (1.1x)	1.49 (1.8x)	2.18 (2.5x)	2.88 (3.3x)
Performer	0.37 (2.3x)	0.59 (4.5x)	0.82 (6.7x)	1.06 (8.9x)

Table 3.5: Per device TPU Peak Memory Usage (GB) in increasing Length input in "Long range Arena"[62]

3.7 Efficient attention comparison

In this work we have decided to test a fixed pattern, a learnable patter and a kernel attention to have confront possible differences between different type of attention. We have chosen inside each category a best representative but it has to be noted that every attention is different and some of the analysis provided cannot simply translate to the attention family of reference. None the less here are the first theoretical differences of chosen attention.
Apart from the obvious type differences we have already discussed in section 3.3, we can understand the main computational complexity differences in Table 3.3. Note that Big Bird seams to have a linear complexity but the clear winner for high values of L.

A good paper to have a first overview into the differences between our attentions is "Long range arena: A benchmark for efficient transformers" [62]. In this paper authors provide a common ground where efficient attention can be tested and compered into with different tasks length. In "Long range arena" datasets chosen to provide increasing challenge for the model, but none of this data is real world data. With the exception of ProbSparse all of our attention are present in this work and according to it's results, displayed in Table 3.4 and Table 3.5, we can see how the Performer is the faster architecture and that the bigbird architecture is only a marginal improvement over original attention. Also memory results are in line, although in this case BigBird is significantly lighter then the Transformer.

We tried to implement similar test with our real world data for time series forecasting.

Chapter 4

Experiments

4.1 Experimental Settings

In this work we try to understand witch of the four attention mechanism performs better resource wise, and if the gain in resource performance is worth the possible drop in accuracy.
We have trained the four variation of the model on two separate multivariate time series datasets to perform forecasting. To recall we will stet the Informer with vanilla attention mechanism(2.4.3), Informer with the so-called ProbSparse attention (3.4), Informer+Performer (3.6), Informer+BigBird (3.5).

In the following section we introduce both datasets and talk about the different challenges they pose on the model. We train the models at different input L and prediction P length. With all the other hyper parameters fixed, this would hopefully give us an understanding of the effects of increasing L (input length) over the computational complexity.
We have carried out test for three pairs of hyper parameter $(L, P) = \{(48; 12),$ $(96; 24),$ $(144; 36)\}$. Batch size has been fixed on $B = 32$. We have divided each dataset into 90% train dataset, 10% validation dataset and 10% test dataset.

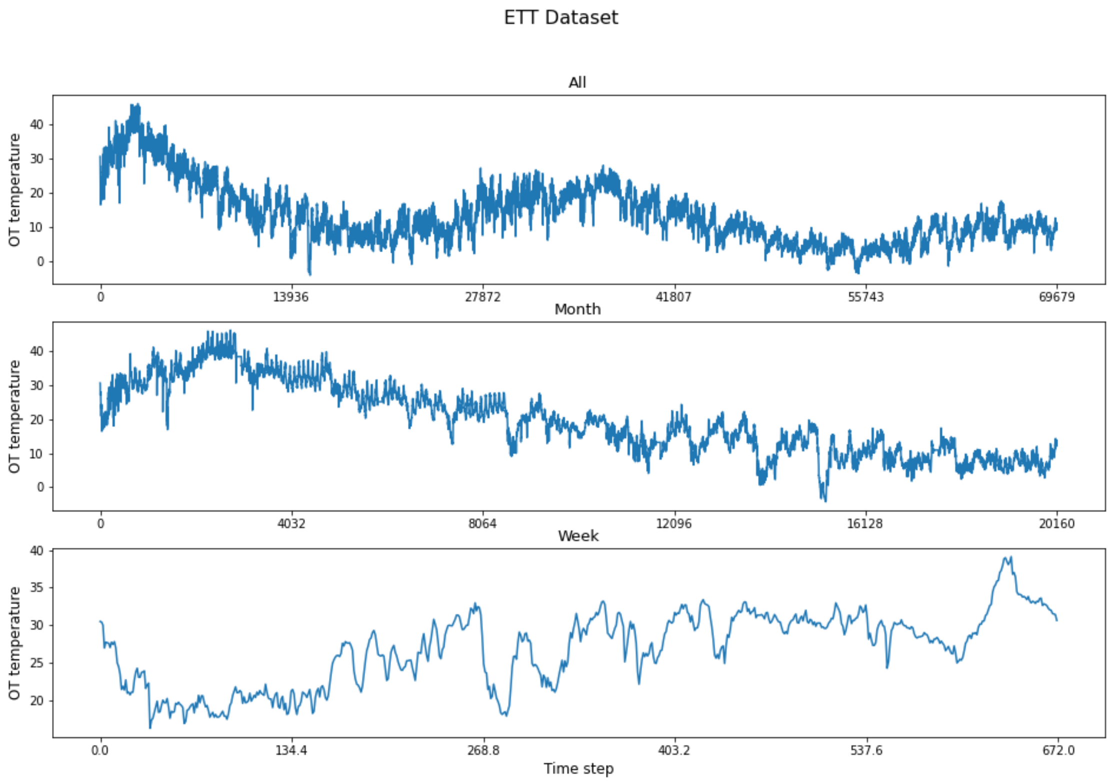

Figure 4.1: ETTm1 Dataset; From top to bottom (all dataset, a month, a week)[5]

4.2 Datasets

We have chosen two different datasets. The first Electricity Transformer Dataset [5] that monitors the temperature of an electric transformer, a measure where variation on the value are slow with respect to the time step dimension. The second CU-BEMS [63] instead with a high variation, it describe the power consumption of an office building. Both dataset have various application possibilities, for what concerns forecasting we can use the resulting model to perform energy optimization or outlier detection. We will now go in detail into the characteristics of every dataset.

4.2.1 Electricity Transformer Dataset

Electrical transformers are not highly responsive machines, they need time, as any major electrical grid component, to switch from high power output to low

power output. Nowadays electric transformer managers have to make decisions, whether to increase power output or not, based on empirical numbers, which are much higher than real-world demands. This leads to wasting electricity as machines work in not ideal condition and reduce the lifespan of transformers. The stakes are high as any false prophecy of demand may damage electrical transformers. For this reason Electricity Transformer Dataset has been developed by Beijing Guowang Fuda Science & Technology Development Company to address this problem.
Electricity Transformer Dataset (ETDataset) was first used in Informer, in its small variant, consists in 2 years of data of 2 Electricity Transformers at 2 stations. For each station, we have a 15 minute timestep and 1 hour timestep versions resulting in four sub datasets: ETTh1, ETTh2, ETTm1 and ETTm2. In this thesis we have only used ETTm1. The target variable is oil temperature of the electrical transformer and it is used to express transformer working condition.
Each of the 69680 datapoint has 8 feature:

- Oil temperature (target)
- Date
- High Use Full Load
- High Use Less Load
- Middle Use Full Load
- Middle Use Less Load
- Low Use Full Load
- Low Use Less Load

Specifically, the dataset combines short-term periodical patterns, long-term periodical patterns, long-term trends, and many irregular patterns as we can see from the Figure 4.1. Note that dataset represents a physical measure so there is no presence of abrupt changes in value, more precisely there is a high local time autocorrelation, this means a more easy problem to solve for the model.

The dataset present also high correlation between input features seen in Figure 4.2. In Table 4.1 we can see an example of a few data points.

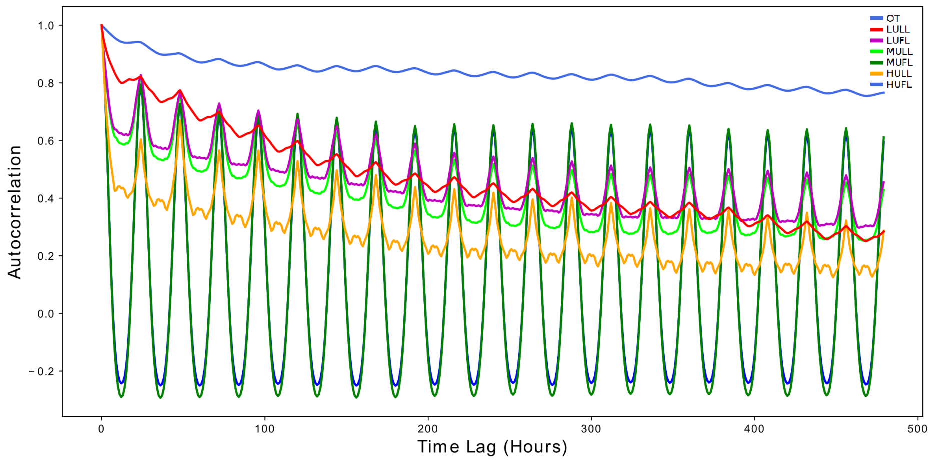

Figure 4.2: ETTm1 feature autocorrelation[5]

date	HUFL	HULL	MUFL	MULL	LUFL	LULL	OT
2016-07-01 00:00	5.827	2.009	1.599	0.462	4.203	1.340	30.531
2016-07-01 00:15	5.760	2.076	1.492	0.426	4.264	1.401	30.460
2016-07-01 00:30	5.760	1.942	1.492	0.391	4.234	1.310	30.038
2016-07-01 00:45	5.760	1.942	1.492	0.426	4.234	1.310	27.013
2016-07-01 01:00	5.693	2.076	1.492	0.426	4.142	1.371	27.787

Table 4.1: ETT datapoints

4.2.2 CU-BEMS, smart building energy and IAQ data

CU-BEMS stands for *Chulalongkorn University Building Energy Management System* [63]. It a dataset that collects various sensors measurement of a university office building in Bangkok, Thailand.
As the complex is a 7 floor construction, the dataset is split into 7 sub datasets one for each floor. Each floor is then divided in zone, see Figure 4.3 for reference, and each zone has its own set of measurement regarding electrical power load and room environment.

Each zone has a feature for:

- Air Conditioning unit power load
- Total plug power load

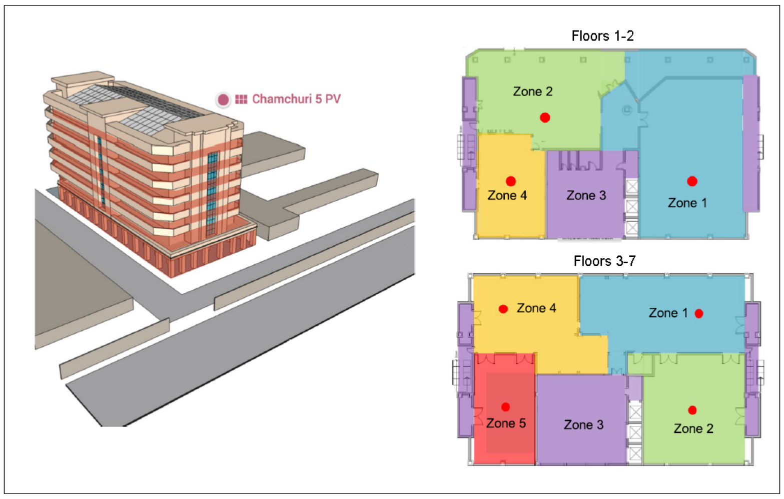

Figure 4.3: Complex render image and zone division [63]

- Lighting power load
- Indoor temperature
- Relative humidity
- Light intensity measured in lux

This means that as for any floor may have different number of air conditioning units, the number of features varies for each sub dataset. The data points have one minute granularity, with 790.560 points to covers a span of total 1,5 years of measurements, from 01/07/2018 to 01/01/2020.
In particular in this thesis we will use the second floor dataset, that has 4 zones with a total of 39 features. The reason for choosing this particular floor is that it has the least amount of missing values.
We also performed some particular preprocessing on the dataset, first we undersample the dataset at 15 minute to uniform it to the ETT Dataset and the Informer [5] original development condition. We also added as a feature the external temperature of Bangkok taken from a local weather unit near the building using public available dataset [63]. And in the end we computed the total power consumption of the floor as the sum of all the power loads, this will be our target variable for the forecasting task.

We want to point out that this dataset presents some diverse challenges to the model, with respect to ETT [5]. As this time it is not a physical measure, but a

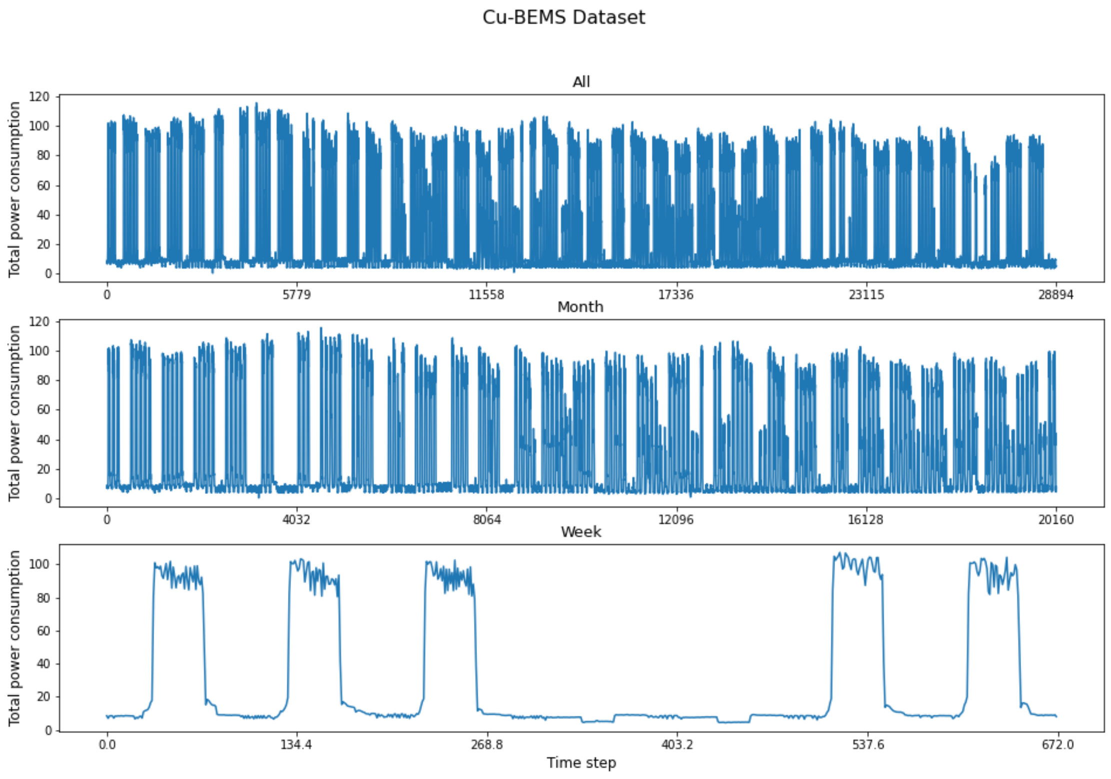

Figure 4.4: CU-BEMS dataset; From top to bottom (all dataset, a month, a week) [63]

human activity, is more subject to abrupt changes and its seasonality component is stronger.

4.3 Training setup

To perform our analysis we set up a test bench to perform every test with the same hardware, eliminating so the possibility of an unfair advantage.
Our machine has the following characteristics:

- CPU : Intel(R) Core(TM) i7-6850K @ 3.60GHz
- RAM : 64GB

- GPU : Nvidia GeForce GTX TITAN X with 12GB VRAM

We trained each model for 10 epochs and evaluate only the best for each configuration according to validation loss.

4.4 Analysis of results

Attention	ETT					
	Train		Val		Test	
	LOSS	MAE	LOSS	MAE	LOSS	MAE
bigb	$3.4e^{-3}$	$45.6e^{-3}$	$14.2e^{-3}$	$89.9e^{-3}$	$4.03e^{-3}$	$53.0e^{-3}$
full	$3.94e^{-3}$	$49.4e^{-3}$	$24.9e^{-3}$	$1.29e+02e^{-3}$	$12.8e^{-3}$	$99.6e^{-3}$
perf	$3.74e^{-3}$	$47.4e^{-3}$	$14.8e^{-3}$	$96.0e^{-3}$	$4.97e^{-3}$	$56.0e^{-3}$
prob	$3.41e^{-3}$	$45.6e^{-3}$	$8.27e^{-3}$	$69.3e^{-3}$	$2.2e^{-3}$	$34.4e^{-3}$

Table 4.2: ETT performance on test set and resources

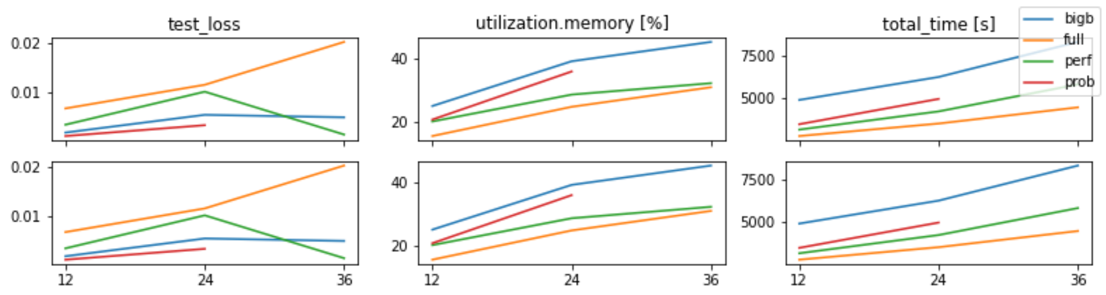

Figure 4.5: ETT results plots

We have carried out test for three pairs of hyper parameter $(L, P) = \{(48; 12), (96; 24), (144; 36)\}$ where L is the input length and P the prediction window length. We recall that the objective of this work is to monitor performance and resources need for each iteration of the model. We note that we weren't able to train the Informer with ProbSparse attention at $(L, P) = (144; 36)$ for an out of memory error.
Talking about performance we can see an example prediction of each model in Figure 4.7 with ETT-Dataset [5] and Figure 4.8 with CU-BEMS [63]. For our task, each efficient attention performance is in line the original attention. This means that at least every efficient attention approximates the real attention mechanism

Attention	CU-BEMS					
	Train		Val		Test	
	LOSS	MAE	LOSS	MAE	LOSS	MAE
bigb	$14.7e^{-3}$	$86.0e^{-3}$	$11.9e^{-3}$	$68.0e^{-3}$	$25.4e^{-3}$	$92.9e^{-3}$
full	$10.1e^{-3}$	$71.2e^{-3}$	$8.29e^{-3}$	$57.7e^{-3}$	$14.3e^{-3}$	$66.6e^{-3}$
perf	$11.3e^{-3}$	$76.0e^{-3}$	$10.2e^{-3}$	$61.6e^{-3}$	$26.6e^{-3}$	$90.9e^{-3}$
prob	$12.2e^{-3}$	$79.1e^{-3}$	$7.95e^{-3}$	$56.2e^{-3}$	$34.9e^{-3}$	$11.1e^{-3}$

Table 4.3: CU-BEMS performance on test set and resources

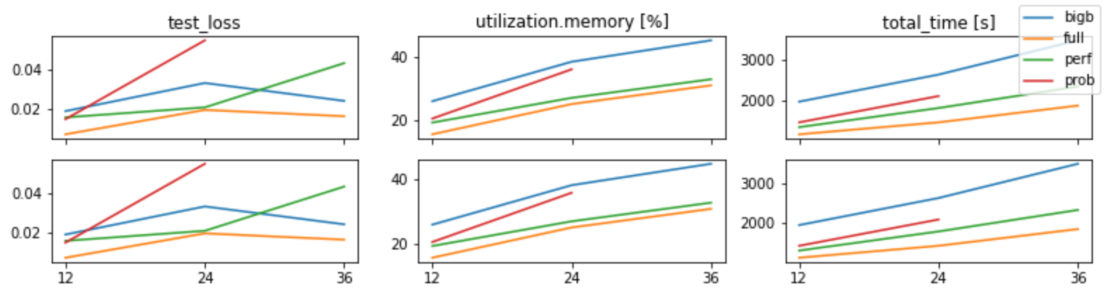

Figure 4.6: CU-BEMS results plots

well enough to have similar results. In some cases as we have seen in section 3.7 the biases of an efficient attention can increase the performance, but for our analysis this is not the case.

To measure used resources we recorded in software wall clock training time and inference time, and through Nvidia query software (nvidia-smi) the GPU load and memory usage.

Looking at the GPU metrics we can appreciate that the full attention performs better than any other efficient variation, only the performer so similarly light weighted. This could sound counter intuitive recalling the theoretical background seen in Chapter 3, but we want to reiterate that any efficient attention come with an overhead cost to compute all the necessary elements to perform the approximation. For this reason some authors of efficient attention like BigBird [4] provided us with an estimated crossing point performance wise. We have already seen the crossing point of some attention according to "Long range arena" [62] in table 3.4 and 3.5.

To better exploit the correlation power of the transformer architecture, we tried to concatenate the input feature, this as also the benefit to stress test our attentions as the input length drastically increases. To accommodate from this increase we also decrease the model hidden dimension representation as it would be an unfair advantage with respect to the previous models. This however can't make up for the out of memory problem of ProbSparse that we will discuss later.

Attention	ETT					
	Train		Val		Test	
	LOSS	MAE	LOSS	MAE	LOSS	MAE
full	$3.9e^{-03}$	$53.2e^{-03}$	$9.0e^{-03}$	$68.0e^{-03}$	$10.3e^{-03}$	$76.9e^{-03}$
bigB	$5.0e^{-03}$	$46.4e^{-03}$	$8.4e^{-03}$	$69.2e^{-03}$	$9.2e^{-03}$	$73.7e^{-03}$
perf	$3.7e^{-03}$	$45.2e^{-03}$	$8.3e^{-03}$	$66.1e^{-03}$	$11.2e^{-03}$	$75.1e^{-03}$

Table 4.4: ETT performance on test set and resources in concatenate setting

Attention	VRAM [MB]	training[min]	inference[s]
Full	7084.1	101.2	19.0
Big Bird	5880.1	151.1	33.8
Performer	6451.8	114.2	24.5

Table 4.5: ETT performance on test set and resources

Due to hardware limitations we could perform the concatenate test with the ETT dataset and the results can be found in table 4.4 and 4.5. As we can see, the overall time is increased with also a loss in performance so this is not advisable practice for the Informer architecture. But nonetheless we can draw some conclusion. First we can see how the efficient attention are now use fare less memory then the full counterpart. Secondly, as we have seen we couldn't complete the training procedure of some model for out of memory error. This also seems to indicate that efficient attention are tailored for high performance machine and high input length while standard attention results more useful for a typical low scale application.

A little footnote, as we have seen the ProbSparce [5] attention uses a lot of memory this is far superior also with respect to the full attention [1], from a theoretical point of view we have already seen that this should not be the case, but taking the code implementation from the original paper of Informer [5] we can see how the authors construct a matrix of dimension $L \times L \times m \times d$ where m is the number of the top queries selected according to Equation 3.3. This is an extreme increase in memory needs for a "efficient attention".

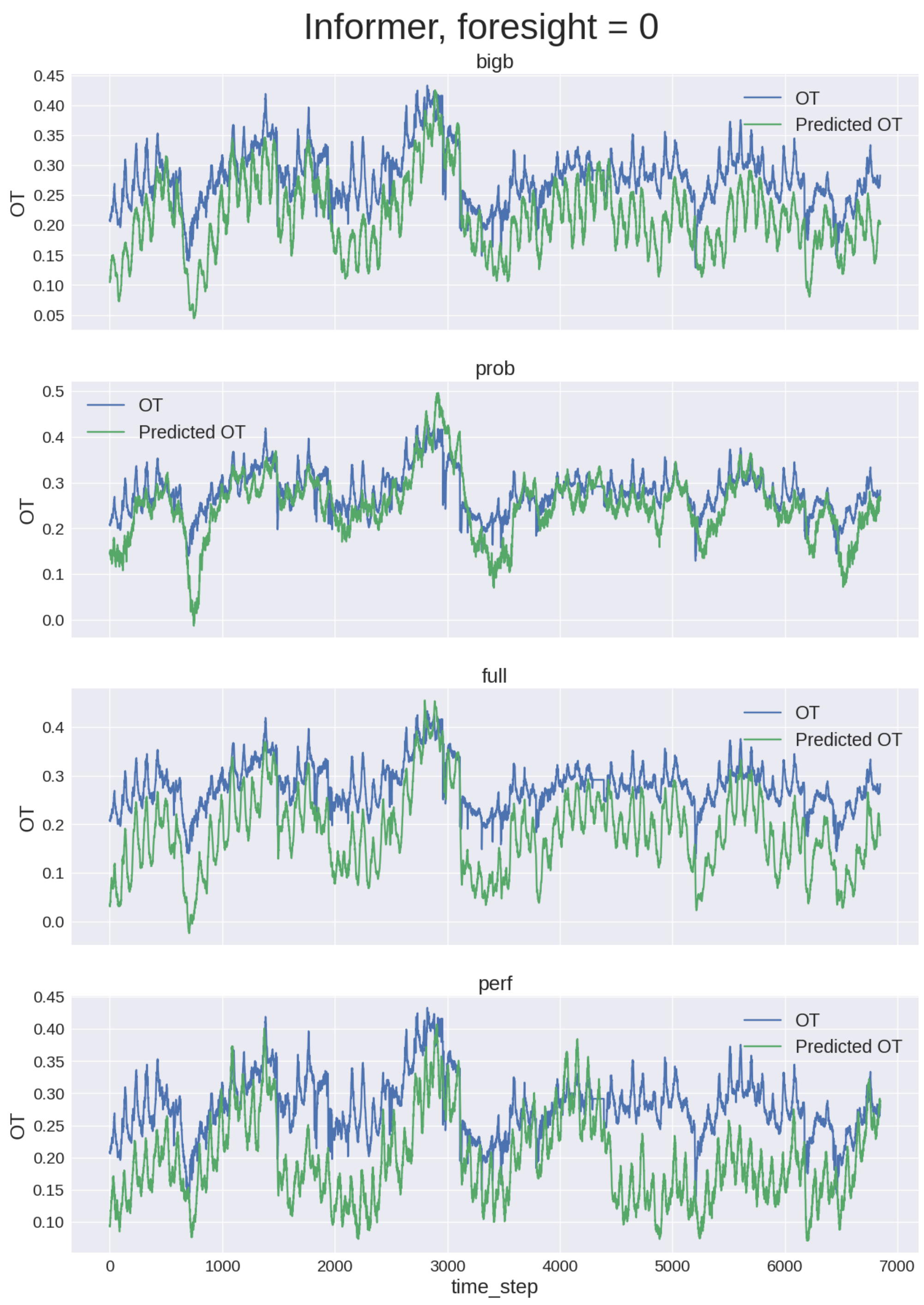

Figure 4.7: Comparison on predictions over the test set(ETT) between various attention with input length $L = 96$

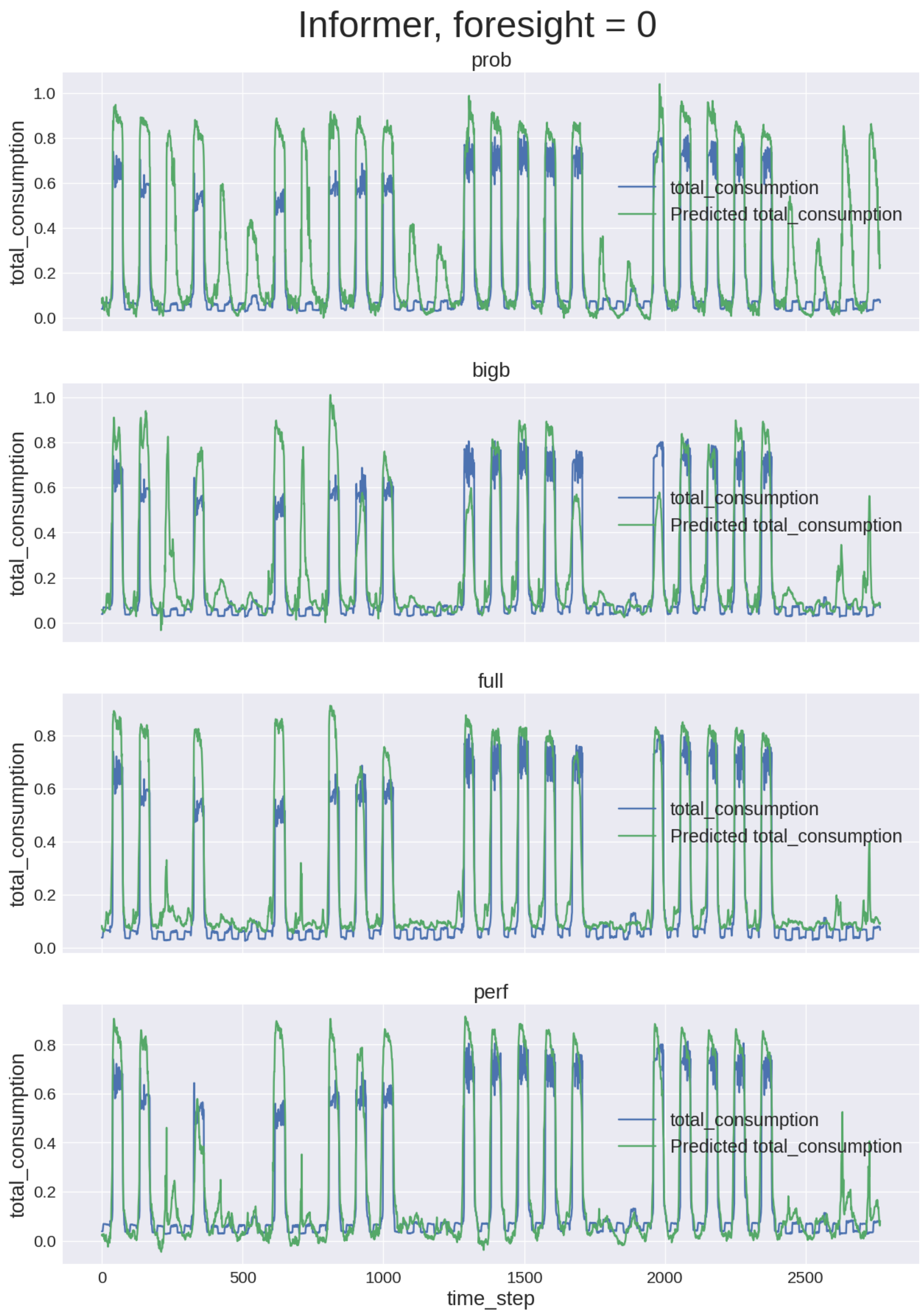

Figure 4.8: Comparison on predictions over the test set(CU-BEMS) between various attention with input length $L = 96$

Chapter 5

Conclusions

Transformer architectures have revolutionized the natural languages processing field and are now approaching other areas of the deep learning research, so much that the interest for more efficient architectures is rising. But although this improvements should decrease hardware requirements for Transformers, this might not be always the case.
This thesis focused on efficient Transformer architectures with application to time series forecasting. In particular, three efficient attention mechanisms introduced in the Informer, the BigBird and the Performer architectures are studied and compared to the attention computed by the vanilla Transformer. The goal is to evaluate whether the use of efficient attention mechanisms could be beneficial for industrial applications. To this end, these attention mechanisms are integrated in the Informer architecture and the four resulting models are trained and tested on publicly available datasets.
The performed experiments indicate that for small data regimes and low specs machine, the use of efficient transformer architectures is not justifiable as they increase the resources needed without any benefit in performance. In fact, many efficient architecture loose the quadratic complexity problem introducing more complex algorithms to calculate the attention approximation, but in doing so they increase the overhead memory cost. Thus, the obtained results suggest that any efficient Transformer architecture modification has to be carefully chosen according to the task and dataset characteristics, as the use of these efficient models might not result in significant benefits in terms of computational resources needed both in training and in inference.
In the future, we could attempt this type of analysis for other efficient attentions and other datasets, with longer sequences, to validate and strengthen the conclusions

proposed here.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, \ Lukasz Kaiser, and Illia Polosukhin. «Attention is All you Need». In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf` (cit. on pp. 1, 2, 32–36, 38, 39, 42, 45, 63).

[2] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. «Learning representations by back-propagating errors». In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: `10.1038/323533a0`. URL: `https://doi.org/10.1038/323533a0` (cit. on pp. 1, 24, 28, 32).

[3] Krzysztof Choromanski et al. «Rethinking attention with performers». In: *arXiv preprint arXiv:2009.14794* (2020) (cit. on pp. 1, 43, 44, 47, 50–52).

[4] Manzil Zaheer et al. «Big bird: Transformers for longer sequences». In: *Advances in Neural Information Processing Systems* 33 (2020) (cit. on pp. 1, 43, 44, 47, 49, 50, 62).

[5] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wan Zhang. «Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting». In: *AAAI*. 2021 (cit. on pp. 1, 39, 40, 47–49, 56, 58, 59, 61, 63).

[6] Athanasopoulos G. Hyndman R.J. *Forecasting: principles and practice.* 3rd. Melbourne, Australia: OTexts, 2021. Chap. 2 (cit. on p. 3).

[7] Athanasopoulos G. Hyndman R.J. *Forecasting: principles and practice.* 3rd. Melbourne, Australia: OTexts, 2021. Chap. 3 (cit. on pp. 5, 6).

[8] Cleveland R. B., Cleveland W. S., McRae J. E., and Terpenning I. J. «STL: A seasonal-trend decomposition procedure based on loess.» In: *Journal of Official Statistics* 6 (1990), pp. 3–33 (cit. on p. 7).

[9] Athanasopoulos G. Hyndman R.J. *Forecasting: principles and practice.* 3rd. Melbourne, Australia: OTexts, 2021. Chap. 7 (cit. on p. 7).

[10] Mikaela Pisani. *TODS: Detecting Different Types of Outliers from Time Series Data.* 2021. URL: `https://towardsdatascience.com/tods-detecting-outliers-from-time-series-data-2d4bd2e91381` (cit. on p. 10).

[11] Sumeyra Demir, Krystof Mincev, Koen Kok, and Nikolaos G. Paterakis. «Data augmentation for time series regression: Applying transformations, autoencoders and adversarial networks to electricity price forecasting». In: *Applied Energy* 304 (2021), p. 117695. ISSN: 0306-2619. DOI: `https://doi.org/10.1016/j.apenergy.2021.117695`. URL: `https://www.sciencedirect.com/science/article/pii/S0306261921010527` (cit. on p. 10).

[12] Athanasopoulos G. Hyndman R.J. *Forecasting: principles and practice.* 3rd. Melbourne, Australia: OTexts, 2021. Chap. 9 (cit. on pp. 12, 14).

[13] Davis R. A. Brockwell P. J. *Introduction to time series and forecasting.* 3rd. Springer, 2016 (cit. on p. 15).

[14] Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell, eds. *Principles of Neural Science.* Third. New York: Elsevier, 1991 (cit. on p. 16).

[15] Egm4313.s12 (Prof. Loc Vu-Quoc). *Neuron3.* 2018. URL: `https://commons.wikimedia.org/wiki/File:Neuron3.png` (cit. on p. 17).

[16] Ethem Alpaydin. *Introduction to Machine Learning.* 3rd. The MIT Press, 2020. Chap. 11 (cit. on pp. 18, 22).

[17] Ethem Alpaydin. *Introduction to Machine Learning.* 3rd. The MIT Press, 2020. Chap. 11 (cit. on pp. 19, 20).

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press, 2016. Chap. 8 (cit. on pp. 22–24, 27, 41).

[19] Ethem Alpaydin. *Introduction to Machine Learning.* 3rd. The MIT Press, 2020. Chap. 11 (cit. on p. 23).

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press, 2016. Chap. 5 (cit. on p. 23).

[21] Jimmy Ba Diederik P. Kingma. «Adam: A Method for Stochastic Optimization». In: *ICLR* (2015) (cit. on p. 23).

[22] Y. T. Zhou and Rama Chellappa. «Computation of optical flow using a neural network». In: *IEEE 1988 International Conference on Neural Networks* (1988), 71–78 vol.2 (cit. on p. 25).

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press, 2016. Chap. 8 (cit. on p. 26).

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep Residual Learning for Image Recognition». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90 (cit. on pp. 26, 27).

[25] Sergey Ioffe and Christian Szegedy. «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift». In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 448–456. URL: https://proceedings.mlr.press/v37/ioffe15.html (cit. on p. 27).

[26] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016. Chap. 10 (cit. on pp. 29, 31, 37).

[27] Sepp Hochreiter and Jürgen Schmidhuber. «Long Short-term Memory». In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735 (cit. on p. 30).

[28] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. «Hybrid speech recognition with Deep Bidirectional LSTM». In: *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*. 2013, pp. 273–278. DOI: 10.1109/ASRU.2013.6707742 (cit. on p. 30).

[29] Qinkun Xiao, Xin Chang, Xue Zhang, and Xing Liu. «Multi-Information Spatial–Temporal LSTM Fusion Continuous Sign Language Neural Machine Translation». In: *IEEE Access* 8 (2020), pp. 216718–216728. DOI: 10.1109/ACCESS.2020.3039539 (cit. on p. 30).

[30] Niek Tax, Ilya Verenich, Marcello La Rosa, and Marlon Dumas. «Predictive Business Process Monitoring with LSTM Neural Networks». In: *Advanced Information Systems Engineering*. Ed. by Eric Dubois and Klaus Pohl. Cham: Springer International Publishing, 2017, pp. 477–492. ISBN: 978-3-319-59536-8 (cit. on p. 30).

[31] Bryan Lim and Stefan Zohren. «Time-series forecasting with deep learning: a survey». In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 379.2194 (2021), p. 20200209. DOI: 10.1098/rsta.2020.0209. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2020.0209. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2020.0209 (cit. on p. 32).

[32] Slawek Smyl. «A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting». In: *International Journal of Forecasting* 36.1 (2020). M4 Competition, pp. 75–85. ISSN: 0169-2070. DOI: `https://doi.org/10.1016/j.ijforecast.2019.03.017`. URL: `https://www.sciencedirect.com/science/article/pii/S0169207019301153` (cit. on p. 32).

[33] Robert G. Brown, Richard F. Meyer, and D. A. D'Esopo. «The Fundamental Theorem of Exponential Smoothing». In: *Operations Research* 9.5 (1961), pp. 673–687. ISSN: 0030364X, 15265463. URL: `http://www.jstor.org/stable/166814` (cit. on p. 32).

[34] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. «Image Transformer». In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, July 2018, pp. 4055–4064. URL: `https://proceedings.mlr.press/v80/parmar18a.html` (cit. on pp. 32, 43, 44).

[35] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Curtis Hawthorne, Andrew M Dai, Matthew D Hoffman, and Douglas Eck. «Music Transformer: Generating Music with Long-Term Structure». In: *arXiv preprint arXiv:1809.04281* (2018) (cit. on pp. 32, 39).

[36] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. «Sequence to Sequence Learning with Neural Networks». In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, pp. 3104–3112 (cit. on pp. 32, 37, 38).

[37] Shubhadeep Roychowdhury. *Word Level English to Marathi Neural Machine Translation using Encoder-Decoder Model*. Feb. 2019. URL: `https://towardsdatascience.com/word-level-english-to-marathi-neural-machine-translation-using-seq2seq-encoder-decoder-lstm-model-1a913f2dc4a7` (cit. on p. 34).

[38] Thang Luong, Hieu Pham, and Christopher D. Manning. «Effective Approaches to Attention-based Neural Machine Translation». In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sept. 2015, pp. 1412–1421. DOI: `10.18653/v1/D15-1166`. URL: `https://aclanthology.org/D15-1166` (cit. on pp. 33, 38).

[39] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. «Efficient Estimation of Word Representations in Vector Space». In: *CoRR* abs/1301.3781 (2013). URL: `http://dblp.uni-trier.de/db/journals/corr/corr1301.html#abs-1301-3781` (cit. on pp. 34, 39).

[40] Sachin Mehta, Marjan Ghazvininejad, Srini Iyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. «DeLighT: Very Deep and Light-weight Transformer». In: *ArXiv* abs/2008.00623 (2020) (cit. on pp. 38, 45).

[41] Markus Freitag and Yaser Al-Onaizan. «Beam Search Strategies for Neural Machine Translation». In: Jan. 2017, pp. 56–60. DOI: `10.18653/v1/W17-3207` (cit. on p. 39).

[42] Bryan Lim, Sercan Ö. Arık, Nicolas Loeff, and Tomas Pfister. «Temporal Fusion Transformers for interpretable multi-horizon time series forecasting». In: *International Journal of Forecasting* 37.4 (2021), pp. 1748–1764. ISSN: 0169-2070. DOI: `https://doi.org/10.1016/j.ijforecast.2021.03.012`. URL: `https://www.sciencedirect.com/science/article/pii/S0169207021000637` (cit. on p. 39).

[43] Seyed Mehran Kazemi et al. «Time2Vec: Learning a Vector Representation of Time». In: *CoRR* abs/1907.05321 (2019). arXiv: `1907.05321`. URL: `http://arxiv.org/abs/1907.05321` (cit. on pp. 39, 40).

[44] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 1st. USA: Cambridge University Press, 2009. Chap. 1. ISBN: 0521424267 (cit. on p. 41).

[45] Itamar Turner-Trauring. *Where's your bottleneck? CPU time vs wallclock time*. Oct. 2021. URL: `https://pythonspeed.com/articles/blocking-cpu-or-io/` (cit. on p. 42).

[46] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. «Imagenet: A large-scale hierarchical image database». In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255 (cit. on p. 42).

[47] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. «Efficient transformers: A survey». In: *arXiv preprint arXiv:2009.06732* (2020) (cit. on p. 43).

[48] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. «Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks». In: *Proceedings of the 36th International Conference on Machine Learning*. 2019, pp. 3744–3753 (cit. on pp. 43, 44).

[49] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. «Generating Long Sequences with Sparse Transformers». In: *URL https://openai.com/blog/sparse-transformers* (2019) (cit. on pp. 43, 44).

[50] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. «Reformer: The efficient transformer». In: *arXiv preprint arXiv:2001.04451* (2020) (cit. on pp. 43, 44).

[51] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. «Efficient content-based sparse attention with routing transformers». In: *Transactions of the Association for Computational Linguistics* 9 (2021), pp. 53–68 (cit. on pp. 43, 44).

[52] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. «Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention». In: *Proceedings of the International Conference on Machine Learning (ICML)*. 2020 (cit. on pp. 43, 44).

[53] Yinhan Liu et al. «Roberta: A robustly optimized bert pretraining approach». In: *arXiv preprint arXiv:1907.11692* (2019) (cit. on p. 45).

[54] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E. Gonzalez. «Train Large, Then Compress: Rethinking Model Size for Efficient Training and Inference of Transformers». In: *CoRR* abs/2002.11794 (2020). arXiv: `2002.11794`. URL: `https://arxiv.org/abs/2002.11794` (cit. on pp. 45, 46).

[55] Zizheng Pan, Bohan Zhuang, Jing Liu, Haoyu He, and Jianfei Cai. «Scalable Vision Transformers With Hierarchical Pooling». In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2021, pp. 377–386 (cit. on p. 45).

[56] Zihang Dai, Guokun Lai, Yiming Yang, and Quoc Le. «Funnel-Transformer: Filtering out Sequential Redundancy for Efficient Language Processing». In: June 2020 (cit. on p. 45).

[57] Young Jin Kim and Hany Hassan Awadalla. «FastFormers: Highly Efficient Transformer Models for Natural Language Understanding». In: *CoRR* abs/2010.13382 (2020). arXiv: `2010.13382`. URL: `https://arxiv.org/abs/2010.13382` (cit. on p. 46).

[58] Mitchell Gordon, Kevin Duh, and Nicholas Andrews. «Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning». In: Jan. 2020, pp. 143–155. DOI: `10.18653/v1/2020.repl4nlp-1.18` (cit. on p. 46).

[59] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. «TinyBERT: Distilling BERT for Natural Language Understanding». In: Sept. 2019 (cit. on p. 46).

[60] Ali Rahimi and Benjamin Recht. «Random Features for Large-Scale Kernel Machines». In: *Advances in Neural Information Processing Systems*. Ed. by J. Platt, D. Koller, Y. Singer, and S. Roweis. Vol. 20. Curran Associates, Inc., 2007. URL: `https://proceedings.neurips.cc/paper/2007/file/013a006f03dbc5392effeb8f18fda755-Paper.pdf` (cit. on p. 52).

[61] Krzysztof M Choromanski, Mark Rowland, and Adrian Weller. «The Unreasonable Effectiveness of Structured Random Orthogonal Embeddings». In: *Advances in Neural Information Processing Systems.* Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: `https://proceedings.neurips.cc/paper/2017/file/bf8229696f7a3bb4700cfddef19fa23f-Paper.pdf` (cit. on p. 52).

[62] Yi Tay et al. «Long range arena: A benchmark for efficient transformers». In: *arXiv preprint arXiv:2011.04006* (2020) (cit. on pp. 53, 62).

[63] Manisa Pipattanasomporn, Gopal Chitalia, Jitkomut Songsiri, Chaodit Aswakul, Wanchalerm Pora, Surapong Suwankawin, Kulyos Audomvongseree, and Naebboon Hoonchareon. «CU-BEMS, smart building electricity consumption and indoor environmental sensor datasets». In: *Scientific Data* 7.1 (July 2020), p. 241. ISSN: 2052-4463. DOI: `10.1038/s41597-020-00582-3`. URL: `https://doi.org/10.1038/s41597-020-00582-3` (cit. on pp. 56, 58–61).