FH University of Applied Sciences TECHNIKUM WIEN

BACHELOR PAPER

Term paper submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Information and Communication Systems and Services

MULTIVARIATE TIME SERIES PREDICTION USING TRANSFORMERS ARCHITECTURE

By: Sergio Tallo Torres

Student Number: ic19b047

Supervisor: Dietmar Millinger

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand, nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Wien, April 28, 2022

Abstract

In December 2017 the advent of the transformer architecture became the most promising solution for Natural Language Processing (NLP) tasks and has since become state of the art for these tasks. However, in NLP tasks, words are encoded to vectors in a multidimensional space, and if sentences are a sequence of words, every task dealing with sequences of multidimensional vectors should be, in theory, suitable to have a solution with a transformer model.
In this work the author will explain the problem that inspire the transformer's research and the details of its architecture in an easy way, will perform a data analysis in a certain data set, and will show a model that, using transformers over a sequence of multidimensional data, can forecast which should be the next element of that sequence. With this forecast any user could, in theory, compares it with the real measurement and obtain information about possible anomalies before they happen.

Note of thanks

(TODO: WRITE THE NOTES OF THANK)

Summary

1 INTRODUCTION 6

2 TRANSFORMERS ARCHITECTURE 7

2.1 THE LIMITATIONS OF RECURRENT NEURAL NETWORKS 7

2.1.1 Backpropagation and Vanishing Gradient over Time 8

2.1.1.1 Backpropagation 8

2.1.1.2 Vanishing (or exploding) Gradient over time 9

2.2 ATTENTION IS ALL YOU NEED. TRANSFORMERS ARCHITECTURE 14

2.2.1 Positional encoding 15

2.2.2 Transformer encoder layer 17

2.2.2.1 Multi-Head Attention layer 17

2.2.3 Transformer decoder layer (English check) 24

2.2.4 Last steps of the Transformers model (English check) 27

2.3 KNOWN IMPLEMENTATIONS (ENGLISH CHECK) 27

3 TRANSFORMER FOR A MULTIVARIATE TIME SERIES TASK (ENGLISH CHECK) 30

3.1 NATURE OF THE DATA. DATA ACQUISITION AND PREPARATION. (ENGLISH CHECK) 30

3.1.1 Nature of the data and Data acquisition 30

3.1.2 Data preparation 33

3.1.3 Data analysis 34

3.1.3.1 Scale of the features values 35

3.1.3.2 Distribution of the data 36

3.1.3.3 Correlation between features 38

3.1.3.4 Final conclusions of the data analysis 42

3.2 MODEL IMPLEMENTATION 44

3.2.1 Training and testing 45

3.2.1.1 Preparing the data for training 45

3.2.1.2 Baseline model 48

3.2.1.3 First training of the transformer model 50

3.2.2 Improving the model 54

4 CONCLUSIONS 59

BIBLIOGRAPHY 61

LIST OF ABBREVIATIONS 64

1 Introduction

Malfunctions in machines have a correlation with their electrical functioning, therefore if measurements of any machine's electrical function are performed, it is possible to know if that machine is working correctly. This idea could be extrapolated to a whole factory. If electrical measurements are performed, it would be possible to know if the machines in the factory are working correctly. If there is a change in what should be a normal measure, could be an early sign of malfunction.
But this process arises two problems. One, there is to many different measurements to be understood by a person. And two, sometimes the changes are not suddenly but a soft change over a sequence of measurements. In those cases, it is necessary the aid of a machine to see and understand those changes. And a machine performing that task, should learn to perform it.
The idea behind this project is to find a machine learning model that, giving a sequence of electrical measurements, could predict with high precision the next element of that sequence. To this purpose I choose to use a transformers model.

I divided this thesis in two different parts. In the first part I will explained in detail how is the architecture of a transformer model, how it works and what where the motivation behind its develop. In the second I presented the use case mentioned before, describe the process of data analysis, the process of coding and tuning of the transformer model and the results given after the training. Last, I will show some conclusions.

Since it advent in December 2017, the transformers model revolutionized the world of machine learning, and I think, as a future machine learning engineer, is important to understand how the model works internally, to use it with full knowledge and not as a black box. This is my main motivation in this thesis.

To have a better and easier understanding of this project, code lines are going to be shown. If the reader is interested in the entire code, is available in GitHub free to download: https://github.com/SergioTallo/Bsc_Thesis

2 Transformers architecture

In this chapter I will explain the initial problem that inspired the first research of the transformer architecture, the details of it, and some of the most used implementations.

2.1 The limitations of Recurrent Neural Networks

Neural Networks are a powerful tool of modern Artificial Intelligence. Feed Forward Neural Networks (FFNs) are Universal Forward Approximators, which make them, in theory, capable of learning any arbitrarily complex decision function[1]. But they don't have the capacity to work with series or ordered sequences of data, which limits the functionality and the use cases in which FFNs could be used with success.
It is well known that traditional FFNs have difficulty understanding the concept of context. When working with a series of data, these networks can only understand, and process, one sample of the data at a time. They cannot consider that this sample has a correlation within the samples occurring before, or how the previous samples can affect the outcome of the actual sample.
This happens when, for example, working with natural language processing (NLP), where a single word can have different meanings depending on the rest of the words in a sentence. A situation where context is important.

To solve this problem, David Rumelhart introduced in 1986 the concept of a recurrent neural network (RNN) [2], in which a recurrent layer is added to a traditional FFN to deal with the previous samples in the training, testing and results process.

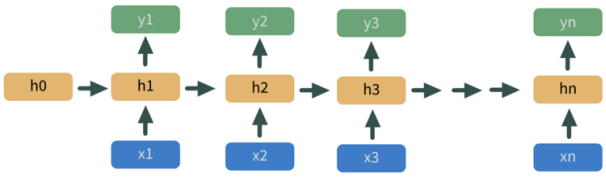


Figure 1: Schematic description of a recurrent neural network[2]

[1] Kurt Hornik, Maxwell Stinchcombe, Halbert White, 'Multilayer feedforward networks are universal approximators',Neural Networks,Volume 2, Issue 5,1989,Pages 359-366,
[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, 'Learning representations by back-propagating errors'.

But, RNNs had to deal with a new problem, the so-called Vanishing Gradient over Time. To explain this concept, I will do an analogy with the way our human memory, especially the short-term memory[3], works.
If I ask the reader of this thesis to tell me which exact word was the first word in it, it would be practically impossible for them to tell without looking it up. The ability to retain exact values in human memory is limited in time, and “vanishes” when we continue to read. In fact, we are dealing with a loss of information when we advance forward in time. In other words, there is a decay of information through time.

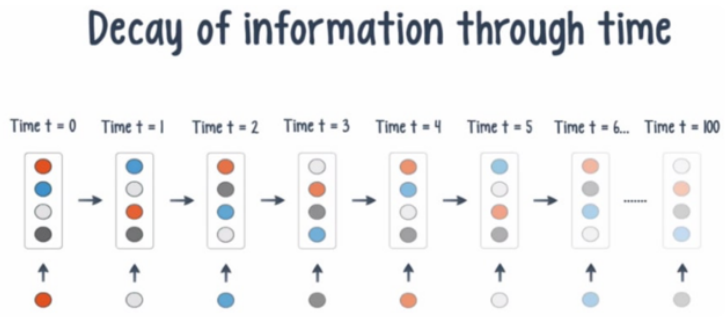


Figure 2: Schematic description of the decay of information[4]

2.1.1 Backpropagation and Vanishing Gradient over Time

The decay of information over time is especially critical in the learning process of a neural network. This process is a backward process. It starts in the last layers of the network and propagates the results to the first layers, using the backpropagation[5] algorithm.

2.1.1.1 Backpropagation

As stated before, the key algorithm making the training and learning process in a neural network possible is the Backpropagation Algorithm. It computes the gradients of all the weights with respect to a given loss function and updates the weights, using gradient descent, to minimize the loss in backward order. It works follows:

---

[3] https://www.simplypsychology.org/short-term-memory.html
[4] https://dev.to/shambhavicodes/let-s-pay-some-attention-33d0
[5] Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (1986a). "Learning representations by back-propagating errors". Nature. 323 (6088): 533–536.

- Compute the delta error in the output layer's units (neurons):

$$\delta s_k = \frac{\partial L}{\partial s_k} = \frac{\partial L}{\partial a_k}\frac{\partial a_k}{\partial s_k} = \frac{\partial L}{\partial a_k} f'(s_k)$$

δ_k : delta error at the k^{th} unit
a_k : activation of the k^{th} unit
s_k : preactivation of the k^{th} unit
L : loss function

- Compute the delta error for the units in the hidden layers

$$\delta_k = \frac{\partial L}{\partial s_k} = \sum_i \frac{\partial L}{\partial s_i}\frac{\partial s_i}{\partial s_k} = \sum_i \delta_i \frac{\partial s_i}{\partial s_k} = \sum_i \delta_i w_{ki} f'(s_i)$$

i : i^{th} unit of the previous layer.
w_{ik} : weight from neuron k to neuron i

- Compute the loss gradients

$$\frac{\partial L}{\partial \delta_{kj}} = \delta_k a_j = \nabla_{k_j}$$

- Update the weights performing gradient descent to minimize the loss function

$$w_{kj}^{new} = w_{kj}^{old} - \eta \nabla_{kj}$$

Now that we know the mathematical explanation of the backpropagation algorithm, can we apply it to explain the vanishing gradient problem.

2.1.1.2 Vanishing (or exploding) Gradient over time

The problem arises when computing the derivative of the activation function. If this derivative is very small, the delta error in this unit will be very small.

In computing the gradient there are three main variables involved:

$$\nabla_{kj} = \delta_k a_j = a_j \sum_i \delta_i w_{ki} f'(s_i)$$

w : weights

$f'(s_i)$: derivative of the activation function

a_j : activation of the previous unit

If the previous unit weight or its activation is small, the importance of this unit in calculating the loss, and therefore in the training, is small. If $f'(s_i)$ is abnormally small (because of the derivative itself), the importance of this unit will be abnormally small. And therefore, the gradient will be also abnormally small.
If the gradient is very small the delta error will be very small, and because the delta error is responsible for the delta errors of the next unit, and therefore also for the gradient, this small gradient will propagate and become even smaller when it reaches the first layers. And therefore, the importance of the first layers will be very small, no matter how the unit's real contribution is.

For example, if we use a sigmoid activation function:

$$f(x) = \frac{1}{1 + e^{-x}}$$

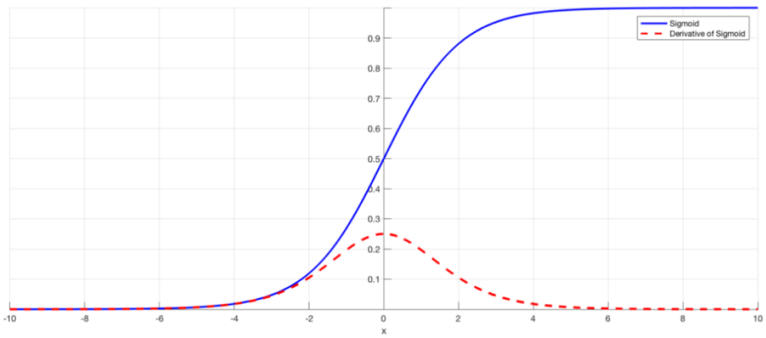


Figure 3: Graph of the sigmoid function and its derivative[6]

[6] https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e

We can notice that the derivative of the sigmoid will returns results that are near to 0, with 0.25 as the maximum possible outcome of a unit in the first hidden layer (recall that when we talk about the backpropagation algorithm, the first hidden layer is the last hidden layer of the network. We are counting backwards). Therefore, in the next layer the maximum value will be 0.0625 (0.25 times 0.25), and in the next 0.0156 (0.0625 times 0.25), and so on. For example, in a neural network with 4 hidden layers, the gradient in the network's first hidden layer would look (in a simplify notation) like this:

$$\vec{\nabla}_1 = \frac{\partial L}{\partial \vec{s_3}} \frac{\partial \vec{s_3}}{\partial \vec{s_2}} \frac{\partial \vec{s_2}}{\partial \vec{s_1}} \frac{\partial \vec{s_1}}{\partial W_1}$$

s_i : vector with the activation functions of the i^{th} layer
W_i: Matrix with the weights of the i^{th} layer
L : Loss function

For a neural network with this architecture:

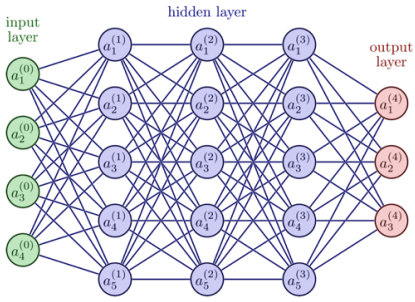


Figure 4: Schema of a small deep neural network[7]

Assuming that the sigmoid function would be used as activation for every unit (except the ones in the output layer), and assuming the squared loss as loss function. The delta error in the units can be calculated as follows:

- Units in the output layer (1 as example):

$$\delta_1^{(4)} = \frac{\partial L}{\partial a_1^{(4)}} f'(s_1^{(4)})$$

[7] https://tikz.net/neural_networks

Assuming the linear function f(x)=x as activation, $f'(s_1^{(4)}) = 1$

Assuming Mean square Loss as loss function and $a_1^{(4)} = \hat{y}_1^{(4)}$.

$$\delta_1^{(4)} = \frac{\partial L}{\partial a_1^{(4)}} = \frac{\partial \frac{1}{2}(y - a_1^{(4)})^2}{\partial a_1^{(4)}} = a_1^{(4)} - y$$

- Units in the third hidden layer (one unit as example)

Assuming the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$ as activation:

$$\delta_1^{(3)} = \frac{\partial \big(sigmoid(a_1^{(3)})\big)}{\partial a_1^{(3)}} \sum_{i=1}^{3} \delta_i^{(4)} w_{1^{(3)} i^{(4)}}$$

Assuming $a_1^{(3)} = 0$ to have max at derivative of sigmoid

$$\delta_1^{(3)} = 0.26 \sum_{i=1}^{3} \delta_i^{(4)} w_{1^{(3)} i^{(4)}}$$

- Units in the second hidden layer (one unit as example and same assumptions as before)

$$\delta_1^{(2)} = 0.25 \sum_{i=0}^{4} \delta_i^{(3)} w_{(1^{(2)} i^{(3)})} = 0.25 \sum_{i=0}^{4} \Big(0.25 \sum_{j=0}^{3} (a_j^{(4)} - y) w_{(1^{(3)} j^{(4)})}\Big) w_{(1^{(2)} i^{(3)})} =$$

$$= 0.25 \cdot 0.25 \sum_{i=0}^{4} \Big(\sum_{j=0}^{3} (a_j^{(4)} - y) w_{(1^{(3)} j^{(4)})}\Big) w_{(1^{(2)} i^{(3)})} = 0.0625 \sum_{i=0}^{4} \Big(\sum_{j=0}^{3} (a_j^{(4)} - y) w_{(1^{(3)} j^{(4)})}\Big) w_{(1^{(2)} i^{(3)})}$$

- Analogously in the first hidden layer (everything same as before):

$$\delta_1^{(1)} = 0.0156 \sum_{l=0}^{4} \Big(\sum_{i=0}^{4} \Big(\sum_{j=0}^{3} (a_j^{(4)} - y) w_{(1^{(3)} j^{(4)})}\Big) w_{(1^{(2)} i^{(3)})}\Big) w_{(1^{(1)} l^{(2)})} =$$

This is the “best case scenario”. In a normal situation, the derivative of the sigmoid function will be even smaller.

If we add more layers, the number will decrease even more, 0.0039 for 5 layers or 0.00097 for 6. Making the vanishing gradient problem bigger.

Recalling that, if the δ-error, and the gradient, vanish over the layers, the gradient descent cannot correctly update the weights and therefore, the learning process will be not efficient, or, in the worst case, not possible.
Because the recurrent layers in a RNN work with the same principle, the vanishing gradient problem will also appear in them, and it is called vanishing gradient over time.

There are some solutions to this problem. The first and easier one is to change the activation function. Using an activation function with a bigger derivative can solve the problem of the smaller gradient.
For example, using a ReLu[8] function:

$$f(x) = x^+ = max(0, x)$$

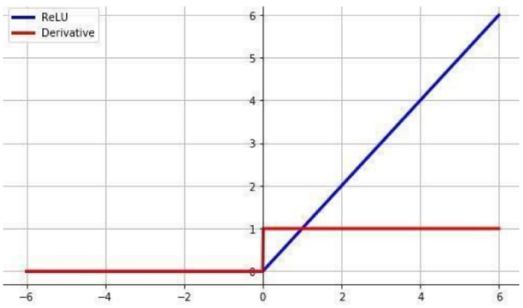


Figure 5: Graph of the ReLu function and its derivative[9]

The derivative is 1 if x=0, therefore the delta error will not be reduced if the activation is small, and therefore the vanishing gradient will be mitigated.

Other approaches are, for example, modified RNNs, like the Long-Short Term Memory (LSTM) unit[10]. Developed by Sepp Hochreiter and Jürgen Schmidhuber in 1997. It tries to solve the vanishing gradient problem letting the gradients pass untouched through the recurrent layers.

---

[8] Fukushima, K. (1969). "Visual feature extraction by a multilayered network of analog threshold elements". IEEE Transactions on Systems Science and Cybernetics.
[9] Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks, Tomasz Szandała. https://arxiv.org/pdf/2010.09458.pdf
[10] Sepp Hochreiter; Jürgen Schmidhuber (1997). "Long short-term memory". Neural Computation. 9 (8): 1735–1780. doi:10.1162/neco.1997.9.8.1735. PMID 9377276. S2CID 1915014.

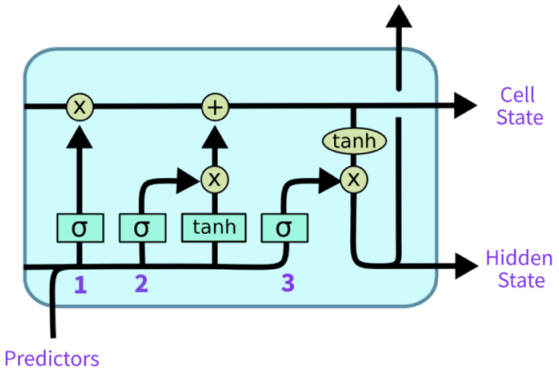


Figure 6: Schematic description of a LSTM unit[11]

Those approaches do not solve the vanishing gradient problem entirely. The one only mitigates it, and the other doesn't consider that the gradient could vanish but also could explode, i.e., the gradient becomes abnormally big across the layers. Therefore, the main problem remains.

Another big problem of RNNs is that they can't parallelize the tasks because they work sequentially. To start one of the sequence samples, the samples before have to be finished.

2.2 Attention is all you need. Transformers Architecture

What if we were able to see a series, not sequentially, but all at once. What if, instead of reading words of a text sequentially, we were able to comprehend the whole text at once. We wouldn't forget the first words of the text and we could see the relations of each word, not only with the previous words, but also with the next ones thus understanding the whole context at once, not sequentially. This would also allow us to use parallelization, which will speed up the whole process substantially. The key to all of this is attention. Providing a concept of attention in the form of an attention mechanism is not something new. There are other models that use this concept of attention in combination with e.g., recurrence or convolution[12].

Using this concept of attention, in December of 2017, a Team at Google Brain[13] presented a paper explaining a new machine learning model proposing that, attention itself is enough to

[11] https://towardsdatascience.comow-to-learn-long-term-trends-with-lstm-c992d32d73be

[12] https://ora.ox.ac.uk/objects/uuid:dd8473bd-2d70-424d-881b-86d9c9c66b51

[13] https://research.google/teams/brain/

learn the context of a series-based data set called “Attention is all you need”[14], by Vaswani et al.

The transformer model is based in a encoder- decoder architecture. With input and output embeddings, positional encoders, multi-head attention modules and feed forward neural networks.

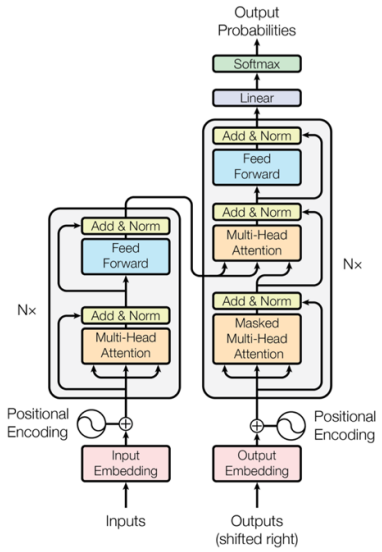


Figure 7: Schematic description of the transformer architecture[13]

In my use case input and output embeddings are not going to be used. Generally, they are used to tokenize the inputs and outputs and to convert them into vectors. The samples in my data set that are already vectors. Therefore, this step is not important for this thesis, but it would be a very important step when working, for example, with NLP projects.

2.2.1 Positional encoding

Even when looking at an entire text at once, instead of sequentially, the individual position of every element in relation to the others is important. For example, take the sentence “A dog is eating a cat.”, using the same words but in a different sequence the meaning could be very different, e.g., “A cat is eating a dog.” or “Is a dog eating a cat?”. Therefore, the model must implement a way to deal with the individual positions of the samples.
The way the transformer model deals with this is by adding to every sample in the sequence another value representing the position of this specific sample in the sequence. We could

[14] https://arxiv.org/abs/1706.03762

therefore say the information of the position of every element is stored in the element itself and not in the structure of the model. A good analogy is the way royalty is named. Instead of having a book telling which Edward was before the other, a number is attached to the name. Therefore, we have Edward the first, Edward the second and so on. Only by looking at the name (the data) we can know the position in history (the sequence).

$$\underset{n\times d}{\text{Input}} = \underset{n\times d}{\text{X}} + \underset{n\times d}{\text{P}}$$

X = Input matrix

P = Positional matrix

n : sequence length

d : number of dimension (features)

The positional matrix is calculated in relation to the sin or cos of the position number follows:

$$\forall pos \in \mathbb{N} | pos \leq n \text{ and } \forall i \in \mathbb{N} | i \leq \frac{d}{2}$$

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

Figure 8: formulas to compute positional encodings[15]

For example, when dealing with a sequence of 3 elements of 4 features, the positional matrix will look like this:

$$P = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0.84 & 0.54 & 0.01 & 1 \\ 0.91 & -0.42 & 0.02 & 1 \end{pmatrix}$$

As stated before, the position matrix will be added to the original input matrix to create a new input matrix including values having the information about the position of their samples.

[15] https://arxiv.org/pdf/1706.03762.pdf

$$X^{new} = \begin{pmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{pmatrix} + \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0.84 & 0.54 & 0.01 & 1 \\ 0.91 & -0.42 & 0.02 & 1 \end{pmatrix}$$

```python
# positional encoding
def positional_encoding(seq_len: int, dim_model: int, device: device = device("cpu")) -> tensor:

    pos = torch.arange(seq_len, dtype=float32, device=device).reshape(1, -1, 1)
    dim = torch.arange(dim_model, dtype=float32, device=device).reshape(1, 1, -1)
    phase = pos / (1e4 ** (torch.div(dim, dim_model, rounding_mode='floor')))

    position_encoding = torch.where(dim.long() % 2 == 0, sin(phase), cos(phase))

    return position_encoding
```

2.2.2 Transformer encoder layer

A transformer model can have one or more encoder layers. Their function is to output a d-dimensional vector representing the transformation of the original data into data including information of the relation between each element in the input sequence.

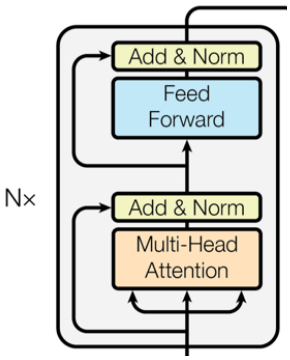


Figure 9: Schematic description of the encoder layer[16]

The encoder layer consists of two main components, one or more multi-head attention blocks and a feed forward neural network. Each having a normalization layer following.

2.2.2.1 Multi-Head Attention layer

The multi-Head attention layer's main purpose is to determine which part of the input it must focus on. It performs a process of self-attention, that is, attention to the relations within itself.

[16] https://arxiv.org/pdf/1706.03762.pdf

The main question to answer is how relevant every element of the sequence is with respect to every other element in the whole sequence. It generates a matrix, composed of one attention vector per element of the sequence, representing the relation of that element to the rest of the elements of the sequence. With this layer the transformer learns the relations between the different elements in the data sequences, such as, words in a sentence.

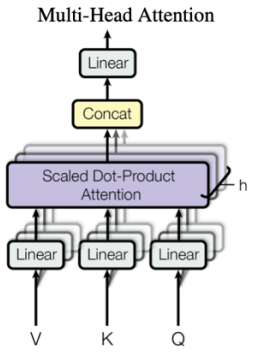


Figure 10: Schematic description of the multi-head attention[17]

For example, working with NLP, we can have as input a sentence like this:

The boy is playing with his brother

The self-attention mechanism will compare every word with every other word in the sentence, compute the (numerical) relation that both words have within each other and store them in a matrix.
If we take the previous sentence, we could have as result a matrix like this (results are not real, they are chosen to show a bigger or smaller relation):

	The	boy	Is	playing	with	his	brother
The	1.00	0.80	0.01	0.02	0.05	0.04	0.03
boy	0.80	1.00	0.90	0.75	0.70	0.90	0.80
is	0.01	0.90	1.00	0.80	0.03	0.02	0.01
playing	0.02	0.75	0.80	1.00	0.50	0.04	0.75
with	0.05	0.80	0.03	0.50	1.00	0.65	0.70
his	0.04	0.90	0.02	0.04	0.65	1.00	0.95
brother	0.03	0.80	0.01	0.75	0.70	0.95	1.00

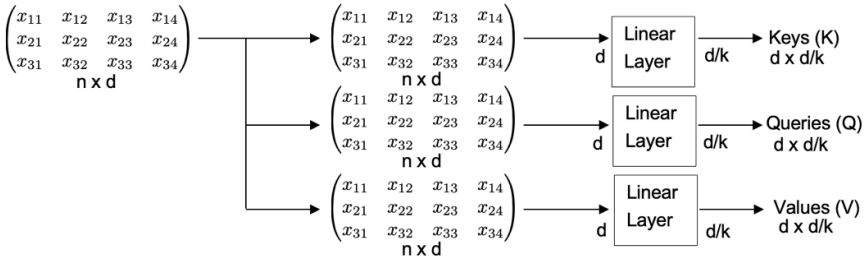
[17] https://arxiv.org/pdf/1706.03762.pdf

In the matrix we can see that “the” has a big relationship with “boy” (it is its article) but a small relationship with the rest of the words. “Boy” has a big relationship also with “is” (verb) and with “brother”. This matrix translates into numbers the relations of the words in that sentence.

Internally, the multi-head attention layer works as follows:

The input matrix (with the positional encoding) is used as input for three different parallel linear layers[18]. They have the number of data features as the number of inputs and the number of features divided by the number of heads (floor division) as the number of outputs. Three different matrices will be computed in these linear layers:

- Keys
- Queries
- Values



k = number of heads

These vectors will be used to calculate the self-attention matrix using the scaled dot-product attention.

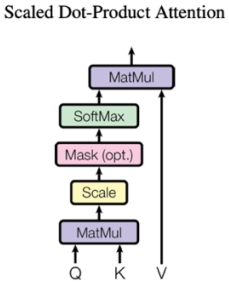


Figure 11: Schematic description of the Scaled Dot-Product Attention[19]

[18] https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

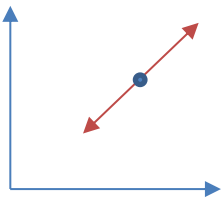
[19] https://arxiv.org/pdf/1706.03762.pdf

It works as follows:

To simplify, suppose an architecture with one head. In the previous step, a linear layer computed a query matrix (2 dimensions, sequence length x number of features) and a key matrix (same dimensions).
The task of the self-attention mechanism is to find the relation of every sample in the sequence with every other sample in the same sequence (and to each other). For this we use the output matrices Queries and Keys. Both come from the same input sequence; therefore, every sample is a transformation of the same original matrix. Therefore, this relation is based on the similarity of every sample transformed in the Queries matrix and every sample transformed in the Keys matrix.
Since every sample is a vector of one dimension, the mathematical explanation for this similarity is based on their direction. If two vectors are similar, their directions in a vector space will be similar. The maximum similarity happens when the vectors are pointing in the same direction ($\vec{v} = \vec{w}$). Otherwise, the minimum similarity happens when the vectors are pointing in the opposite direction ($\vec{v} = -\vec{w}$).

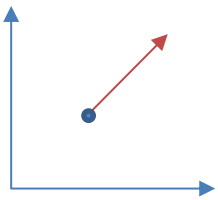
Since a neural network works better with values between 1 and -1, the self-attention mechanism uses the cosine similarity.

Minimal similarity



180°, cos (180) = -1

Maximal similarity



0°, cos (0) = 1

We can derive the similarity from the formula of the Euclidean dot product.

$$\vec{v} \cdot \vec{w} = ||\vec{v}||||\vec{w}||cos(\alpha) \rightarrow cos(\alpha) = \frac{\vec{v} \cdot \vec{w}}{||\vec{v}||||\vec{w}||}$$

And since we are working with matrices, we obtain the following square matrix with these dimensions (sequence length x sequence length)

$$SimilarityMatrix = \frac{QK^T}{\sqrt{\frac{d}{k}}}$$

When the similarity matrix is computed, it could be important to mask the results[20]. This operation is optional but important in many use cases. The encoder's masking task is to transform the attention values to 0 wherever is only padding.
I will discuss the masking task itself when talking about the decoder.
In the next step, the similarity matrix will be passed through a softmax function to transform the existing values into values between 1.00 for the maximum similarity and 0.00 to the minimum similarity.

In a sequence with 3 samples, it could look like this:

$$\begin{pmatrix} 1.00 & 0.32 & 0.21 \\ 0.32 & 1.00 & 0.47 \\ 0.21 & 0.47 & 1.00 \end{pmatrix}$$

And finally, we compute the product between the results of the softmax function and the value matrix resulting from the third linear layer.

$$AttMatrix(Q, K, V) = softmax\Big(\frac{QK^T}{\sqrt{d}}\Big)V$$

An analogy of this process could be the following example. The query linear layer creates a matrix with locks, and the key linear layer creates a matrix with keys. If one key is similar to one lock, it will open easily (the probability of open it will be near 1), if the key is not similar to the lock, it will not open it easily (the probability of not open it will be near 1, therefore the probability of open it will be near -1).

[20] https://towardsdatascience.com/how-to-code-the-transformer-in-pytorch-24db27c8f9ec

```python
def scaled_dot_product_attention(query: tensor, key: tensor, value: tensor, mask: tensor = None) -> tensor:
    # https://pytorch.org/docs/stable/generated/torch.bmm.html
    # https://pytorch.org/docs/stable/generated/torch.transpose.html
    # https://pytorch.org/docs/stable/generated/torch.nn.functional.softmax.html#torch.nn.functional.softmax

    # Step 1: matrix multiplication between query and key transpose
    # Transpose dimension 2 and 3
    key = key.transpose(1, 2)
    step_1 = torch.bmm(query, key)
    # Dimensions after step_ 1 [batch_size, sequence_len, sequence_len]

    # Step 2: Scale the matrix by dividing by the square root of the key dimension (number of features)
    d_k = query.size(-1)
    step_2 = step_1 / math.sqrt(d_k)
    # Dimensions after step_ 2 [batch_size, sequence_len, sequence_len]

    # Step 3: Apply the mask to the attention matrix if mask is not None
    if mask is not None:
        step_3 = step_2 + mask
    else:
        step_3 = step_2

    # Step 4: Apply softmax to the attention matrix
    step_4 = f.softmax(step_3, dim=-1)
    # Dimensions after step_4 [batch_size, sequence_len, sequence_len]

    # Step 5: Multiply the attention matrix with the values
    step_5 = torch.bmm(step_4, value)
    # Dimensions after step_5 [batch_size, sequence_len, number of features]

    return step_5
```

Currently we are supposing only one self-attention head. If there were more than one heads, the results (matrix with 2 dimensions, sequence length x (sequence length / number of heads)) must be appended to each other to a resulting matrix (sequence length x sequence length).

Suppose a sequence length of 4 with 2 heads:

$$concatenate(\underset{4\times2}{SM_{HEAD_1}}, \underset{4\times2}{SM_{HEAD_2}}) = \underset{4\times4}{SM_{FINAL}}$$

The result of the dot scaled dot product attention goes through another linear layer and the output will be added to the original matrix (after the positional encoding) and the resulting matrix will be normalized.

```python
class Residual(nn.Module):

    # https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html
    # https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html

    def __init__(self, sublayer: nn.Module, dimension: int, dropout: float = 0.1):
        super().__init__()
        self.sublayer = sublayer
        self.norm = nn.LayerNorm(dimension)
        self.dropout = nn.Dropout(dropout)

    def forward(self, *tensors: tensor) -> tensor:
        # Assume that the "query" tensor is given first, so we can compute the
        # residual.  This matches the signature of 'MultiHeadAttention'.

        # Step 1: Apply the sublayer
        step_1 = self.dropout(self.sublayer(*tensors))

        # Step 2: Add input and output of sublayer
        step_2 = step_1 + tensors[0]

        # Step 3: Apply the layer norm
        step_3 = self.norm(step_2)

        return step_3
```

The last layer of the encoder layer is a so-called position-wise FFN. This is nothing else as a traditional linear FFN with one hidden layer with a ReLu activation function and dropout operation between them. The standard configuration is 2048 units in the hidden layer.

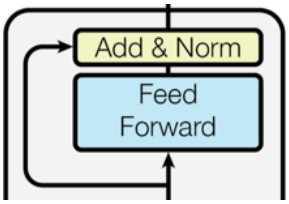


Figure 12: Schematic description of the last FFN in the encoder layer[21]

Finally, to compute the output of the encoder layer, the input of this FFN will be added to the output and after the addition, a normalization process will be performed.

```python
class Residual(nn.Module):

    # https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html
    # https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html

    def __init__(self, sublayer: nn.Module, dimension: int, dropout: float = 0.1):
        super().__init__()
        self.sublayer = sublayer
        self.norm = nn.LayerNorm(dimension)
        self.dropout = nn.Dropout(dropout)

    def forward(self, *tensors: tensor) -> tensor:
        # Assume that the "query" tensor is given first, so we can compute the
        # residual.  This matches the signature of 'MultiHeadAttention'.

        # Step 1: Apply the sublayer
        step_1 = self.dropout(self.sublayer(*tensors))

        # Step 2: Add input and output of sublayer
        step_2 = step_1 + tensors[0]

        # Step 3: Apply the layer norm
        step_3 = self.norm(step_2)

        return step_3
```

In the training process the backpropagation algorithm will update the weights of every linear layer in every encoder layer starting from the last FFN till the linear layers that compute the Queries, Keys and Values matrices.

Suppose an architecture with two encoder layers each of them with two multi-head attention layers, the delta errors and therefore, the gradients, will propagate as follows:

[21] https://arxiv.org/pdf/1706.03762.pdf

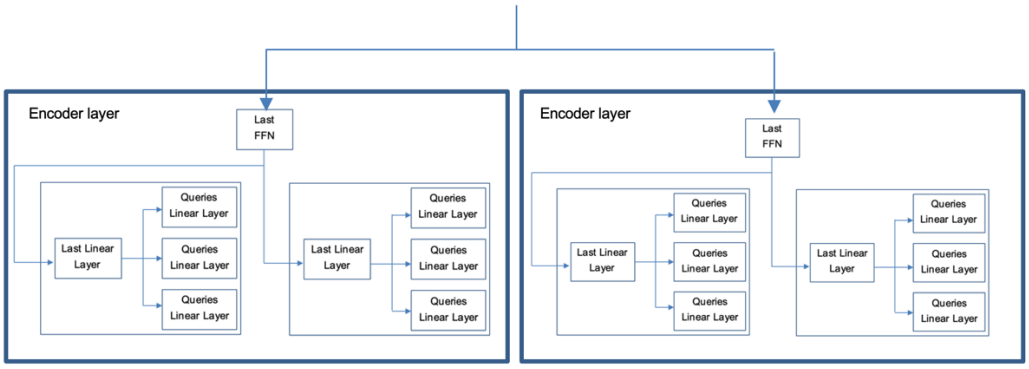
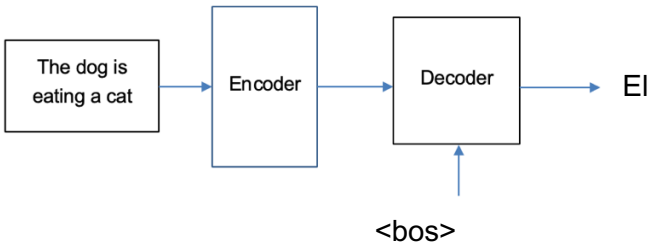


Figure 13: Schematic description of the backpropagation algorithm in the encoder layer

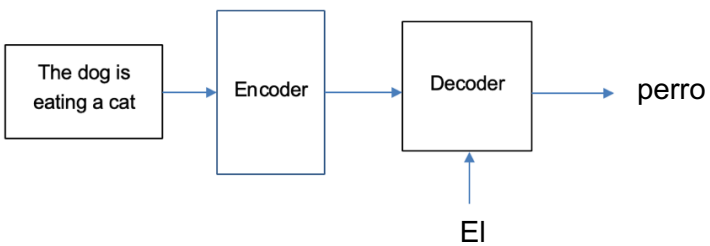
2.2.3 Transformer decoder layer

Same as in the encoder layer, a transformer model can have one or more decoder layers. Their function is to output a d-dimensional vector with the result values of the model. In the learning process it needs a target sequence shifted to the right, this means the input of the decoder is the already computed sequence, starting from a “null” position. For example, in an NLP task, the first target would be an empty token, meaning that the sentence starts. When one word is computed, this word will be used as input to compute the second. A sentence composed by both words will be used for the third word, and so on. For example, if the task is to translate the sentence “The dog is eating a cat” to Spanish, the process would look as follows:

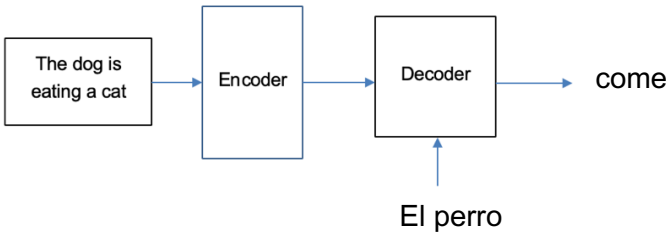
1st Step:



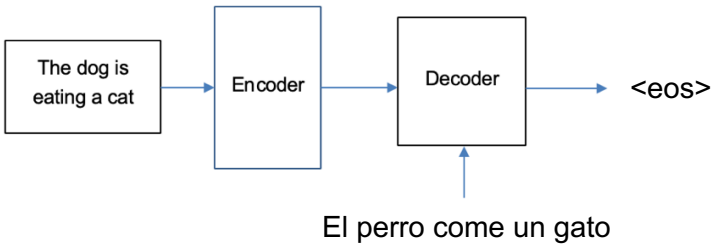
2nd step:



3rd step:



And so on, till the last sequence value.



In detail, the decoder layer works as follow:

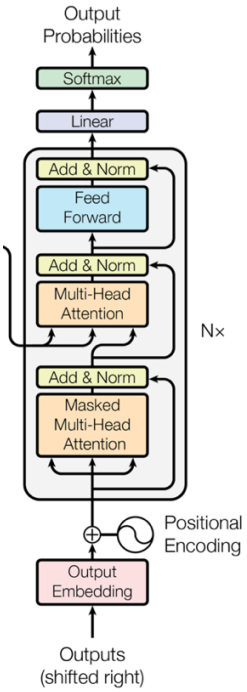


Figure 14: Schematic description of the decoder layer [22]

[22] https://arxiv.org/pdf/1706.03762.pdf

The target sequence goes through the same self-attention mechanism as in the encoder layer, but here the mask has a different task as in the encoder. The masking task in the decoder is to prevent the sequence to look up the samples ahead of the actual sample on the sequence.
The process works as follows:
A masking matrix will be created. Its dimensions are the same as the self-attention matrix, and their values are -infinity in the values above the diagonal, and 0 in the values below the diagonal and the diagonal itself. This matrix will be added to the attention matrix, resulting in a process as follows:

Suppose a self-attention matrix of 2 dimensions (3 x 3):

With these dimensions we generate this matrix:

$$\begin{pmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{pmatrix}$$

We should than add this matrix to the attention matrix resulting of the similarity product between the Queries matrix and the Keys matrix. Resulting in a matrix where all values above the diagonal are -infinty, and the rest of the values remains invariable.

For example, it could look like this:

$$\begin{pmatrix} 1.00 & 0.32 & 0.21 \\ 0.32 & 1.00 & 0.47 \\ 0.21 & 0.47 & 1.00 \end{pmatrix} + \begin{pmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1.00 & -\infty & -\infty \\ 0.32 & 1.00 & -\infty \\ 0.21 & 0.47 & 1.00 \end{pmatrix}$$

The rest of the self-attention mechanism in the decoder is the same as in the encoder.

But, in the decoder there is a second self-attention mechanism, involving the output of the encoder. In this case, the output of the encoder is used to calculate new Keys and Queries matrices, and the matrix resulting from the previous decoder self-attention head is used to calculate the new Values matrix. As an analogy, the output of the encoder is the memory of the past values of a sequence, and the resulting matrix from the decoder self-attention mechanism are the next values of the sequence.

Schematically, it looks like this:

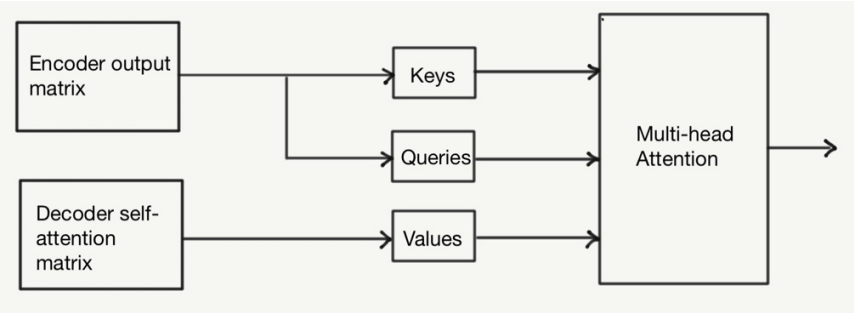


Figure 15: Schematic description of the 2nd Attention layer in the decoder

After this process, the rest of the steps in the decoder layer are the same as in the encoder layer.
As in the encoder layer, there could be multiple multi-head attention layers, and multiple decoder layers. And the backpropagation algorithm, propagates in the same way as in the encoder layer.

2.2.4 Last steps of the Transformers model

After the decoder layers, it only remains to “wrap up” the results of the whole process. Another linear layer will catch the results and their output is treated differently depending on the kind of output we are interested in. For example, if we are dealing with an NLP task, a softmax function will transform the results in probabilities that show which word should be the next one in the sequence. Otherwise, for example, in numerical regression tasks (the one this thesis is dealing with), there is no need the output of the decoder will be the output of the decoder.
The output of the decoder will be used then to compute the next value in the sequence, using it as input in the decoder and repeating the whole process till the output sequence is completed.

2.3 Known implementations

Since the advent of “Attention is all you need” in December 2017, many teams do their own implementations of the transformers model. The most known and used are the PyTorch[23]

[23] https://pytorch.org/hub/huggingface_pytorch-transformers/

and Tensorflow[24] libraries. Many big models based in the transformers model were developed.

Some of them are, for example:

- BERT[25]: Is a language modelling and next sentence prediction transformer model, developed by google in 2018. Follows almost exactly the implementation of the original paper. Is nowadays, one of the most baseline models used for experiments with NLP.

- GPT-3[26]: The third generation of the GPT series, is a pretrained transformer developed for NLP tasks. Has a capacity of 175 billion parameters, and was trained with more than 500 billion tokens.

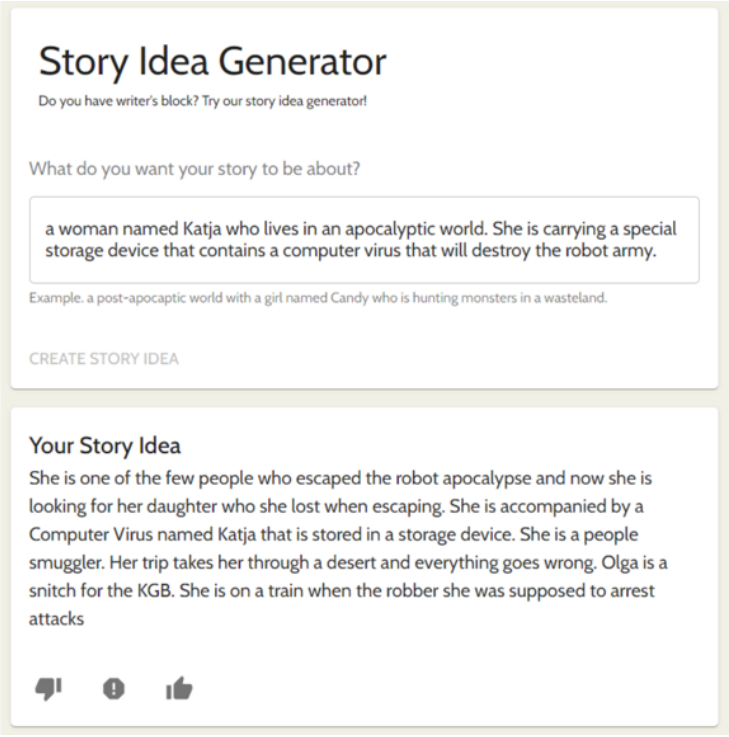


Figure 16: Text generated by GPT-3[27]

[24] https://www.tensorflow.org/text/tutorials/transformer

[25] https://github.com/google-research/bert

[26] https://openai.com/blog/openai-api/

[27] https://www.reddit.com/r/GPT3/comments/k3rboj/a_simple_story_idea_generator_using_gpt3/

- DALL-E[28]: Based on GPT-3, DALL-E is an implementation of a Transformer that interpret natural language inputs to generate corresponding images. It has 12-billion parameters and was trained on 400 million pairs of text and image.

Figure17: Example of images generated by DALL-E[29]

[28] https://openai.com/blog/dall-e/

[29] https://analyticsindiamag.com/openai-to-change-the-digital-image-making-game-with-dall-e-2-its-text-to-image-generator/

3 Transformer for a multivariate time series task

Using a data set composed by electrical measurements from a German Factory, I trained a transformer model that can predict the next element of a sequence of measurements.
In this chapter I will describe the process I followed, starting from the data analysis, and following with the actual develop and coding of the model. I will show the results that I obtained, and some improvements based on the model first results.

3.1 Nature of the data. Data acquisition and preparation.

Any learning process, no matter if it is a human or a machine process is always based on data. A child learning to differentiate a dog from a cat, need to see an actual cat or dog (or pictures of the) many times till is able to do it. Same if it is a computer vision algorithm. Therefore, data is the most important part of any machine learning model, and the first process that any machine learning engineer should do, is to know, analyze and understand the available data.
The data I am using in this project are electrical measurements of a factory in Germany. Which factory it is, where is located and what is the kind of machines that are being used there is, for the purposes of this thesis, irrelevant.

3.1.1 Nature of the data and Data acquisition

The dataset is composed of 18 different features (19 if we count the timestamp). All of them are electrical measurements obtained by IOT sensors and register in a database.
For this thesis I received the data already registered in a .csv file. I don't have any further information about the process of obtaining the information, neither how the data collection process works, nor how the data is sent and stored in the database.
In this project I am using 18 different measures, described with the name contained in the .csv file, as follows:

- Current Power: “The Electric power is the rate, per unit time, at which electrical energy is transferred by an electric circuit”[30]. Measured in Watts.
  In this case the measure reflects the instant electrical power in one of the three phases.
  PLN1: This is the current power measured in phase number 1. In Watts (W)
  PLN2: This is the current power measured in phase number 2. In Watts (W)
  PLN3: This is the current power measured in phase number 3. In Watts (W)

$$P = \frac{W}{t} = \frac{W}{Q}\frac{Q}{t} = VI$$

  Q : Electric charge (in Coulombs)
  t : Time in seconds (in seconds)
  I : Electric current (in Amperes)
  V : Voltage (in Volts)

- Current Voltage: Often used as a potential difference. It is the difference of the potential energy between two points of a circuit[31].
  In this case, the measure reflects the difference between one of the three phases and the neutral wire.
  ULL1: Current Voltage between phase 1 and neutral wire. In Volts (V)
  ULL2: Current Voltage between phase 2 and neutral wire. In Volts (V)
  ULL3: Current Voltage between phase 3 and neutral wire. In Volts (V)

$$\nabla V_{AB} = V(r_B) - V(r_A)$$

  $V(r_B)$: Voltage (in Volts) measured in the point B
  $V(r_A)$: Voltage (in Volts) measured in the point a

[30] https://www.electronics-notes.com/articles/basic_concepts/power/what-is-electrical-power-basics-tutorial.php
[31] https://www.fluke.com/en/learn/blog/electrical/what-is-voltage

- Power Factor: “Is the relationship (phase) of current and voltage in AC electrical distribution systems”[32]. In other words, it is the measure on how effective the electricity is used in our circuit. Ideally should be 1.
  COS_PHI1: Power Factor in phase number 1.
  COS_PHI2: Power Factor in phase number 2.
  COS_PHI3: Power Factor in phase number 3.

- Frequency of the alternate current: Is the times per second that the electrical signal alternates[33], measured in Herz, in Europe is 50Hz.
  FREQ: Frequency of the alternate electrical current. In Hertz (Hz)

$$f = \frac{1}{T}$$

  T : Period in seconds

- Fault current: Is the electrical current that flows out of a circuit in the event of one or more of the conductors shorting each other[34].
  RC_DC: Fault current in direct current. In Volts (V)
  RC_AC: Fault current in alternate current. In Volts (V)
  RC_50Hz: Fault current in alternate current. In Volts (V)
  RC_150Hz: Fault current in alternate current. In Volts (V)
  RC_<100Hz: Fault current in alternate current. In Volts (V)
  RC_100Hz – 1KHz: Fault current in alternate current. In Volts (V)
  RC_>100KHz: Fault current in alternate current. In Volts (V)

Every minute, the sensors take one measurement and send the data to the database.
In the whole data set I received for this project, there are 63360 samples each with 18 measurements and one timestamp.

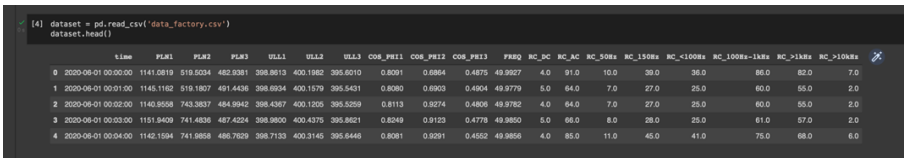
[32] https://www.laurenselectric.com/home/business/understanding-power-factor/
[33] https://www.fluke.com/en/learn/blog/electrical/what-is-frequency
[34] https://www.elandcables.com/the-cable-lab/faqs/faq-what-is-fault-current

3.1.2 Data preparation

To use the raw data is not always possible. The data could be in a not readable format, or it could have some values that do not fit in the set. That could make it difficult to manipulate. Therefore, normally it needs some form of preparation.

To manipulate, analyze and use the data that I received in the .csv file, I used the pandas' package[35], currently is known to be as the state of the arte package to work with data frames and big amounts of data in python[36]

First, I needed to parse the whole .csv file and load the data into a data frame for further preparation.



```
[4] dataset = pd.read_csv('data_factory.csv')
    dataset.head()
```

	time	PLN1	PLN2	PLN3	ULL1	ULL2	ULL3	COS_PHI1	COS_PHI2	COS_PHI3	FREQ	RC_DC	RC_AC	RC_50Hz	RC_150Hz	RC_<100Hz	RC_100Hz-1kHz	RC_>1kHz	RC_>10kHz
0	2020-06-01 00:00:00	1141.0819	519.5034	482.9381	398.8613	400.1982	395.6010	0.8091	0.6864	0.4875	49.9927	4.0	91.0	10.0	39.0	36.0	86.0	82.0	7.0
1	2020-06-01 00:01:00	1145.1162	519.1807	491.4436	398.6934	400.1579	395.5431	0.8080	0.6903	0.4904	49.9779	5.0	64.0	7.0	27.0	25.0	60.0	55.0	2.0
2	2020-06-01 00:02:00	1140.9558	743.3837	484.9942	398.4367	400.1205	395.5259	0.8113	0.9274	0.4806	49.9782	4.0	64.0	7.0	27.0	25.0	60.0	55.0	2.0
3	2020-06-01 00:03:00	1151.9409	741.4836	487.4224	398.9800	400.4375	395.8621	0.8249	0.9123	0.4778	49.9850	5.0	66.0	8.0	28.0	25.0	61.0	57.0	2.0
4	2020-06-01 00:04:00	1142.1594	741.9858	486.7629	398.7133	400.3145	395.6446	0.8081	0.9291	0.4552	49.9856	4.0	85.0	11.0	45.0	41.0	75.0	68.0	6.0

Is important to search for the failures during the process of data acquisition. This will be shown in the .csv file as a NaN value. These values could be a problem for the learning model; therefore, they should be replaced properly. There are numerous ways to deal with this problem[37], the most popular are:

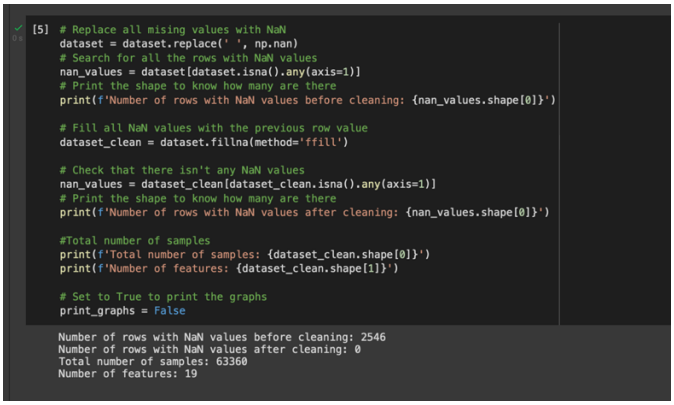
- Replace with zeros: Where the NaN values are replaced by 0.
- Replace with Mean: Where the NaN values are replaced by the mean of all that feature values.
- Replace with Median: Where the NaN values are replaced by the median of all that feature values.
- Replace with previous value: Where the NaN values are replaced by the last correct value before this NaN value.

[35] https://pandas.pydata.org/docs/getting_started/overview.html

[36] https://makemeanalyst.com/data-science-with-python/python-libraries-for-data-analysis/

[37] https://towardsdatascience.com/whats-the-best-way-to-handle-nan-values-62d50f738fc

In this case I replaced the NaN values by the previous value. Normally the electrical data should not change a lot in one minute, therefore, in most situations the incorrect value would most likely be similar to the value one minute ago.
The provided dataset contained 2546 NaN measures (with a total of 45828 NaN values) over 63360 total measures. Equalling 4.02% failed measurements. In my opinion, this is a large number and should be dealt with in the future, but it is not a part of this thesis, therefore I am not going discuss further.
For the task of replacing the NaN values, I used the provided function fillna[38] in the pandas' package. After this process, I had a clean data set with 63360 entries of 19 features (one of the features is the measurement timestamp).



After this process, I had a data frame ready to be analyzed.

3.1.3 Data analysis

Before using the data to train any machine learning model, it is useful to perform an analysis of the data. Machine learning and data analysis go "hand in hand". A good data analysis helps to understand the data we are working with and will help us to choose the correct model and to set the right parameters and hyperparameters[39] of our learning model.

Every minute one measurement is taken and stored; therefore, 1440 samples were taken every day. The whole data set consists of 63360 samples, distributed over 44 days (1440 x

[38] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html
[39] https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/

44 = 63360). The first measure was on the 1st of June of 2020 and the last one on the 14th of July of 2020.

3.1.3.1 Scale of the features values

First, I compute, for every feature, the maximum and minimum value, the median and the mean, and the standard deviation. Doing this we can have a broad information about how wide the data range we are dealing with is.

```
for column in dataset_clean.columns:
  if column == 'time':
    print(column)
    print('Min value: ', dataset_clean[column].min())
    print('Max value: ', dataset_clean[column].max())
    print('')
  else:
    print(column)
    print('Min value: ', dataset_clean[column].min())
    print('Max value: ', dataset_clean[column].max())
    print('Mean value: ', dataset_clean[column].mean())
    print('Median value: ', dataset_clean[column].median())
    print('Standard deviation: ', dataset_clean[column].std())
    print('')
```

I obtained these results:

```
PLN1
Min value:  1136.6154
Max value:  26071.959
Mean value:  7185.2709603472795
Median value:  4370.43185
Standard deviation:  5619.401583329123

PLN2
Min value:  330.2979
Max value:  16792.3887
Mean value:  4645.443629026148
Median value:  4504.5791
Standard deviation:  3948.189347661856

PLN3
Min value:  465.1976
Max value:  14512.8389
Mean value:  4081.8074168496946
Median value:  3310.39745
Standard deviation:  3423.479296987532

ULL1
Min value:  384.115
Max value:  418.8481
Mean value:  398.5690919760104
Median value:  399.0723
Standard deviation:  4.60327571313802

ULL2
Min value:  386.556
Max value:  420.7076
Mean value:  400.6287229008857
Median value:  401.09614999999997
Standard deviation:  4.379149353374775

ULL3
Min value:  384.4315
Max value:  418.0959
Mean value:  398.1920267992438
Median value:  398.7023
Standard deviation:  4.1864513722862595

COS_PHI1
Min value:  0.791
Max value:  1.0
Mean value:  0.9403826672980489
Median value:  0.9874
Standard deviation:  0.0702666167556289

COS_PHI2
Min value:  0.3818
Max value:  1.0
Mean value:  0.9094865893308393
Median value:  0.9854
Standard deviation:  0.12151698460262582
```

```
COS_PHI3
Min value:  0.4249
Max value:  1.0
Mean value:  0.8106457149621302
Median value:  0.961
Standard deviation:  0.21538851292958983

FREQ
Min value:  49.8598
Max value:  50.1182
Mean value:  49.999149831123994
Median value:  49.9993
Standard deviation:  0.01864464532400887

RC_DC
Min value:  1.0
Max value:  11.0
Mean value:  4.546243686868687
Median value:  5.0
Standard deviation:  0.6682858249786129

RC_AC
Min value:  37.0
Max value:  178.0
Mean value:  79.47874053030303
Median value:  80.0
Standard deviation:  18.213949757652767

RC_50Hz
Min value:  3.0
Max value:  14.0
Mean value:  8.381549873737374
Median value:  8.0
Standard deviation:  1.5043985121902832

RC_150Hz
Min value:  18.0
Max value:  47.0
Mean value:  32.317960858585856
Median value:  32.0
Standard deviation:  6.713189268559589

RC_<100Hz
Min value:  16.0
Max value:  45.0
Mean value:  28.909406565656564
Median value:  28.0
Standard deviation:  6.198981696195879

RC_100Hz-1kHz
Min value:  36.0
Max value:  166.0
Mean value:  74.09142992424242
Median value:  75.0
Standard deviation:  17.14211118810104
```

```
RC_>1kHz
Min value:  26.0
Max value:  172.0
Mean value:  68.76529356060605
Median value:  68.0
Standard deviation:  17.71488225498485

RC_>10kHz
Min value:  0.0
Max value:  11.0
Mean value:  3.77209595959596
Median value:  3.0
Standard deviation:  1.507438092488837
```

With one quick look at these results, I could determine that there is a big difference in the scale of the different features. Some features, like PLN1 are in the scale of ten thousands (between 1000 and 26000) while other, like COS_PHI, are between 0 and 1. That could be a problem for the training of the model and would require to do some sort of normalization and standardization[40].

3.1.3.2Distribution of the data

After that, I computed some plots to visualize the nature of the data and its distribution. I calculated two types of plots for every feature. A line plot with the data of a whole week, and a boxplot for every week.

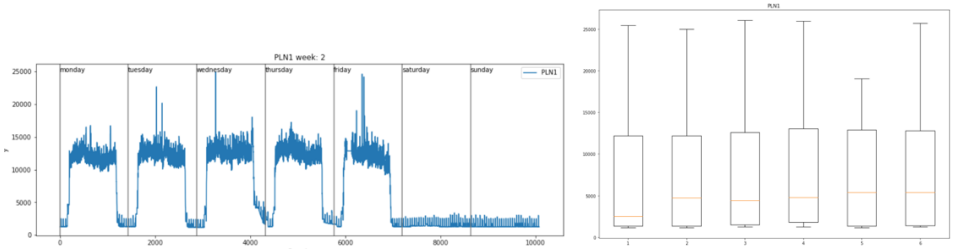
```
# Set to True to print the graphs
print_graphs = True

if print_graphs is True:

  # Iterate to every column to print a graph of every feature
  for i, column in enumerate(dataset_clean.columns):
    if i > 0:
      # Feature in a weekly interval
      utils_bsc.week_plot(dataset_clean, i, column)
      # Feature in a daily interval (only the values of weekdays between 4:00 and 19:30)
      utils_bsc.daily_plot(dataset_clean, i, column)
```

```
# Function to print a graph of the data of one feature in a weekly basis
def week_plot(data, col, name):
    # Every day has 1440 entries (one every minute, 60*24=1440)
    days = [0, 1440, 2880, 4320, 5760, 7200, 8640]
    daysname = ['monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'saturday', 'sunday']

    dataret = []

    j = 0
    count = 1

    for i in range(10080, data.shape[0], 10080):

        dataplot = np.array(data.iloc[j:i, col])

        # Line plot with the data of every week
        plt.figure(figsize=(15, 5))
        plt.plot(np.arange(0, dataplot.shape[0]), dataplot, label=name)

        # Delimitate every day to easy see the dat of every day
        for k in range(len(days)):
            plt.axvline(days[k], color='gray')
            plt.text(days[k] - 5, max(dataplot), daysname[k])

        plt.legend()
        plt.title(f'{name} week: {count}')
        plt.xlabel('Sample')
        plt.ylabel('y')
        plt.show()
        j = i
        count += 1

        dataret.append(dataplot)

    # Print also a boxplot with tghe data of every week
    fig, ax1 = plt.subplots(figsize=(15, 10))
    ax1.set_title(name)
    ax1.boxplot(dataret)
    plt.show()
```

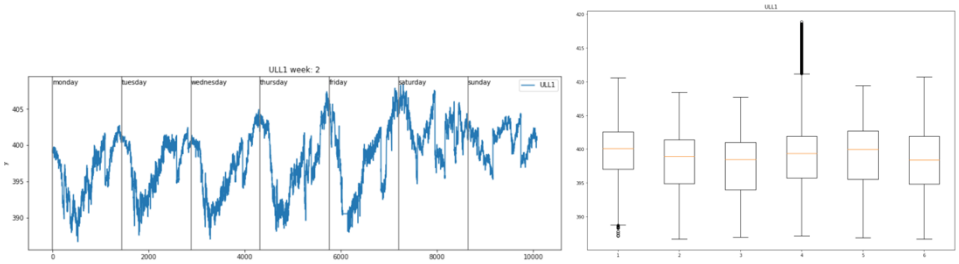
[40] https://towardsdatascience.com/understand-data-normalization-in-machine-learning-8ff3062101f0

The plots of the different features were these:
(For the weekly line plot, I am only going to show one of the weeks. The other weeks have similar distribution)

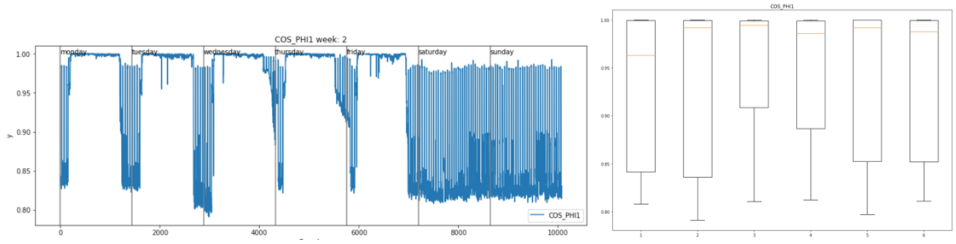
- PLN1, PLN2, PLN3 (The data of these 3 features is very similar, therefore, I am only going to show the plots of PLN):



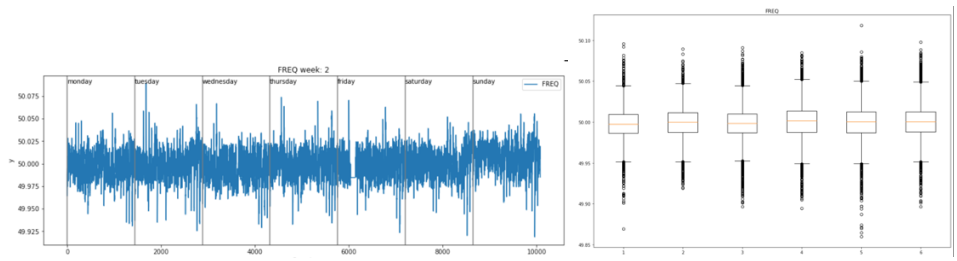
- ULL1, ULL2, ULL3:



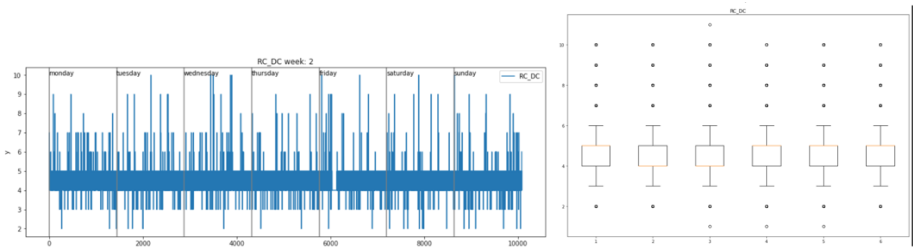
- COS_PHI1, COS_PHI2, COS_PHI3:



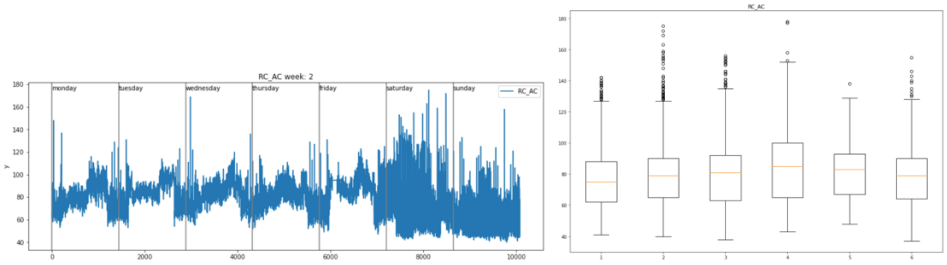
- FREQ



- RC_DC



- RC_AC:



The rest of features (RC RC_50Hz, RC_150Hz, RC_<100Hz, RC_100Hz – 1KHz, RC_>100KHz) are very similar in its distribution, and can be found in the Git repository[41] accompanying this Thesis.

3.1.3.3 Correlation between features

Another important analysis is to determine if there are correlations between the different features. To perform this analysis, I used the spearman-rank correlation coefficient[42].

$$r_s = \rho_{\mathrm{R}(X),\mathrm{R}(Y)} = \frac{\mathrm{cov}(\mathrm{R}(X),\mathrm{R}(Y))}{\sigma_{\mathrm{R}(X)}\sigma_{\mathrm{R}(Y)}}$$

[41] https://github.com/SergioTallo/Bsc_Thesis/blob/master/bsc_arbeit.ipynb

[42] https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient

```
correlations = []

for i in dataset_norm.columns[1:]:
  for j in dataset_norm.columns[1:]:
    print(f'Correlation between {i} and {j}')
    correlation = np.corrcoef(dataset_norm[i], dataset_norm[j])
    if i != j:
      correlations.append(correlation)
      print(correlation[0][1])
  print('')

print(f'Mean of all correlations: {np.mean(correlations)}')
```

I obtained these results:

There is generally a big correlation between the features. The mean of all correlations (17 x 17, 289 different correlations) is 0.457.

Looking at the features individually, we can observe these values:

- PLN1, PLN2 and PLN3 are very similar:
  The mean of the correlations of PLN1 and the rest of the features is: 0.592
  The mean of the correlations of PLN2 and the rest of the features is: 0.580
  The mean of the correlations of PLN3 and the rest of the features is: 0.584
  Between them the correlations are higher than 0.94
  With COS_PHI1 and COS_PHI3, higher that 0.80
  With RC_DC, less than 0.05
  With FREQ, a negative correlation smaller than -0.15

- ULL1, ULL2 and ULL3 are very similar too:
  The mean of the correlations of ULL1 and the rest of the features is: 0.495
  The mean of the correlations of ULL2 and the rest of the features is: 0.480
  The mean of the correlations of ULL3 and the rest of the features is: 0.485
  Between them the correlations are higher than 0.96
  With PLN1, PLN2 and PLN3, the negative correlation is higher than -0.67
  With RC_DC, negative correlation smaller than -0.02
  With FREQ, a negative correlation smaller than -0.13

- COS_PHI1, COS_PHI2, COS_PHI3:
  The mean of the correlations of COS_PHI1 and the rest of the features is: 0.510
  The mean of the correlations of COS_PHI2 and the rest of the features is: 0.434

The mean of the correlations of COS_PHI3 and the rest of the features is: 0.509
Between them the correlations are between 0.67 and 0.77
With PLN1, PLN2 and PLN3, the correlation is higher than 0.80
With RC_DC, negative correlation smaller than 0.03
With FREQ, a negative correlation smaller than -0.15

- FREQ:
  The mean of the correlations of FREQ and the rest of the features is: 0.105
  There is no correlation with FREQ higher than 0.16 (positive or negative)
  The negative correlation with RC_DC is smaller than -0.01

- RC_DC
  The mean of the correlations of RC_DC and the rest of the features is: 0.035
  There is no correlation with RC_CD higher than 0.08 (positive or negative)

- RC_AC, RC_100Hz-1kHz and RC_>1kHz:
  The mean of the correlations of RC_AC and the rest of the features is: 0.545
  The mean of the correlations of RC_AC and the rest of the features is: 0.550
  The mean of the correlations of RC_AC and the rest of the features is: 0.507
  The correlation between them is higher than 0.98
  The correlations with all the PLN are higher than 0.59
  The correlations with RC_DC are 0.016

- RC_50Hz, RC_150Hz, RC_<100Hz:
  The mean of the correlations of RC_50Hz and the rest of the features is: 0.545
  The mean of the correlations of RC_150Hz and the rest of the features is: 0.550
  The mean of the correlations of RC_<100Hz and the rest of the features is: 0.507
  The correlation between them is higher than 0.93
  The correlations with RC_<100Hz are higher than 0.870
  The negative correlations with FREQ are smaller than -0.06
  The negative correlations with RC_DC are smaller than -0.08

- RC_>10kHz:
  The mean of the correlations of RC_>10Hz and the rest of the features is: 0.409

The correlations with RC_50Hz, RC_150Hz or RC_<100Hz are higher than 0.82

The negative correlations with FREQ is -0.044

The negative correlations with RC_DC is -0.077

In general, we can observe that all the features have a very small correlation with FREQ and RC_DC. The rest of the features could be grouped in PLN, ULL, COS_PHI and two groups for RC. This observation matches the logical expectations because those groups of features correspond to similar physical observations.

When working with multidimensional data is useful to perform a variance analysis, computing the covariance matrix of the features, their eigenvalues, and the explained variance of the principal components.

```
# Covariance matrix, eigenvalues and explained variance

covmatrix = dataset_norm.cov()
eigenvalues, eigenvectors = np.linalg.eig(covmatrix)

acc = 0

for i, eigen in enumerate(eigenvalues):
  acc += eigen/np.sum(eigenvalues)
  print(f'Explained_variance {i +1} eigenvalue: {eigen/np.sum(eigenvalues)} (accumulated {round(acc, 4)})')
```

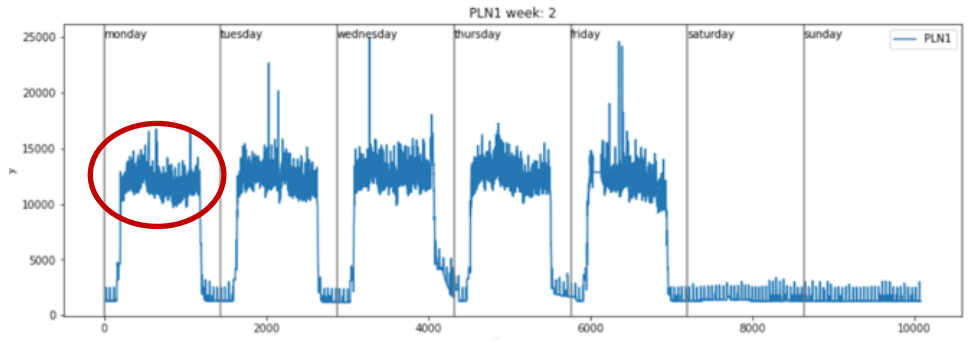
After this computation, I obtained following results:

```
Explained_variance 1 principal component: 0.5317647804810274 (accumulated 0.5318)
Explained_variance 2 principal component: 0.16335739298653476 (accumulated 0.6951)
Explained_variance 3 principal component: 0.07511546472382995 (accumulated 0.7702)
Explained_variance 4 principal component: 0.054921627068028424 (accumulated 0.8252)
Explained_variance 5 principal component: 0.05390616867076577 (accumulated 0.8791)
Explained_variance 6 principal component: 0.04952232661739343 (accumulated 0.9286)
Explained_variance 7 principal component: 0.022486349463995598 (accumulated 0.9511)
Explained_variance 8 principal component: 0.013932731902136385 (accumulated 0.965)
Explained_variance 9 principal component: 0.012792662672300325 (accumulated 0.9778)
Explained_variance 10 principal component: 0.009829012007199104 (accumulated 0.9876)
Explained_variance 11 principal component: 0.004024926426955747 (accumulated 0.9917)
Explained_variance 12 principal component: 0.002893959610103366 (accumulated 0.9945)
Explained_variance 13 principal component: 0.002163271201445878 (accumulated 0.9967)
Explained_variance 14 principal component: 0.001741116222641519 (accumulated 0.9985)
Explained_variance 15 principal component: 0.0006928432299862775 (accumulated 0.9991)
Explained_variance 16 principal component: 0.0002182633742610345 5 (accumulated 0.9994)
Explained_variance 17 principal component: 0.0002335182091708306 4 (accumulated 0.9996)
Explained_variance 18 principal component: 0.0004035851322244031 (accumulated 1.0)
```

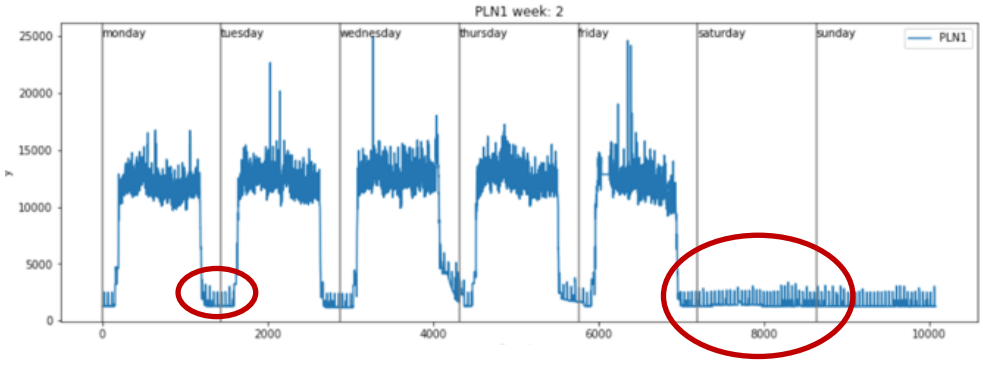
Within the first 6 principal components more of 90% of the variance is explained. Even, within the first 9 principal components (half of the features) more than 97% of the variance is explained.

3.1.3.4 Final conclusions of the data analysis

Immediately, I observed two very different time frames.
The first happened during the weekdays, between 4:30 and 19:30.



The second, occurred during the whole day on weekends and on weekdays between 19:30 and 4:30.



My first intuition was to think that the factory doesn't have any activity during the second time frame. If that is the real situation, I could simply eliminate these samples, because the focus should stay in the working periods and data outside this period could only bring noise into the dataset. But, after a question to my thesis supervisor I found that the factory is not closed, but it has a very different activity. Therefore, I kept using those values as a part of the data frame. Furthermore, further experiments in the early stages of this project showed that there are no

advantages in the results of the model training using the whole set than using only one of those timeframes. Therefore, assuming both reasons mentioned, that first intuition was discarded.

Secondly, the big variation in the scale of the different features will arise a problem in the training process. Therefore, a normalization and standardization will be needed. Of all the different normalization processes, I choose to do a mean standard deviation normalization[43]. The mean of this feature value will be subtracted to the original value and the result will be divided by the standard deviation of this feature.

$$x_{ij}^{new} = \frac{x_{ij}^{old} - \mu_j}{\sqrt{\frac{\sum_i (x_{ij}^{old} - \mu_j)^2}{n}}} = \frac{x_{ij}^{old} - \mu_j}{\sigma_j^2}$$

i : sample index

j : feature index

```python
# apply the mean / stddev scaling in Pandas using the .mean() and .std() methods
def normalize_mean_std_dataset(df):
    # copy the dataframe
    df_norm = df.copy()
    # apply mean / stddev scaling
    for column in tqdm(df_norm.columns):
      if column != 'time':
        df_norm[column] = (df_norm[column] - df_norm[column].mean()) / df_norm[column].std()
    return df_norm
```

After this process the values are scaled in such a way that the mean of all the values from the same feature is 0 and their standard deviation is 1.

Some of the values before the normalization process:

	time	PLN1	PLN2	PLN3	ULL1	ULL2	ULL3	COS_PHI1	COS_PHI2	COS_PHI3	FREQ	RC_DC	RC_AC	RC_50Hz	RC_150Hz	RC_<100Hz
0	2020-06-01 00:00:00	1141.0819	519.5034	482.9381	398.8613	400.1982	395.6010	0.8091	0.6864	0.4875	49.9927	4.0	91.0	10.0	39.0	36.0
1	2020-06-01 00:01:00	1145.1162	519.1807	491.4436	398.6934	400.1579	395.5431	0.8080	0.6903	0.4904	49.9779	5.0	64.0	7.0	27.0	25.0
2	2020-06-01 00:02:00	1140.9558	743.3837	484.9942	398.4367	400.1205	395.5259	0.8113	0.9274	0.4806	49.9782	4.0	64.0	7.0	27.0	25.0
3	2020-06-01 00:03:00	1151.9409	741.4836	487.4224	398.9800	400.4375	395.8621	0.8249	0.9123	0.4778	49.9850	5.0	66.0	8.0	28.0	25.0
4	2020-06-01 00:04:00	1142.1594	741.9858	486.7629	398.7133	400.3145	395.6446	0.8081	0.9291	0.4552	49.9856	4.0	85.0	11.0	45.0	41.0

[43] https://towardsdatascience.com/understand-data-normalization-in-machine-learning-8ff3062101f0

Some of the values after the normalization process:

	time	PLN1	PLN2	PLN3	ULL1	ULL2	ULL3	COS_PHI1	COS_PHI2	COS_PHI3	FREQ	RC_DC	RC_AC	RC_50Hz	RC_150Hz	RC_<100Hz
0	2020-06-01 00:00:00	-1.075593	-1.045021	-1.051232	0.063478	-0.098312	-0.618908	-1.868350	-1.835847	-1.500292	-0.345935	-0.817380	0.632551	1.075812	0.995360	1.143832
1	2020-06-01 00:01:00	-1.074875	-1.045103	-1.048747	0.027004	-0.107515	-0.632738	-1.884005	-1.803753	-1.486828	-1.139728	0.678985	-0.849829	-0.918340	-0.792166	-0.630653
2	2020-06-01 00:02:00	-1.075615	-0.988316	-1.050631	-0.028760	-0.116055	-0.636846	-1.837041	0.147415	-1.532327	-1.123638	-0.817380	-0.849829	-0.918340	-0.792166	-0.630653
3	2020-06-01 00:03:00	-1.073661	-0.988798	-1.049922	0.089264	-0.043667	-0.556540	-1.643493	0.023152	-1.545327	-0.758922	0.678985	-0.740023	-0.253623	-0.643206	-0.630653
4	2020-06-01 00:04:00	-1.075401	-0.988670	-1.050114	0.031327	-0.071754	-0.608493	-1.882582	0.161405	-1.650254	-0.726741	-0.817380	0.303134	1.740530	1.889123	1.950416

The third conclusion is that some of the features are very strongly correlated. A variation of this strong correlation could be a sign that something is not working correctly. It also shows that any technique of dimensionality reduction (for example PCA or t-SNE) could be useful, even some features could be redundant[44]. Nevertheless, we must consider that performing any process of dimensionality reduction will bring loss of information.
In the case of our data set, I don't think any kind of dimensionality reduction or feature selection is needed for the first stages of the training. There are only 18 features, which is a number easy to handle by any computer nowadays, therefore we can train the model with all the features and not lose any information. However, to improve the results of the training or to achieve new insights, revisiting the idea that the most of the variance is within the first 6 principal components could be useful in the future

3.2 Model implementation

Now that we have a good understanding of the data, it is time to implement the model and the training process that will allow us to forecast the next element of a sequence of 30 measures, having 30 consecutive measures and a new element as input.

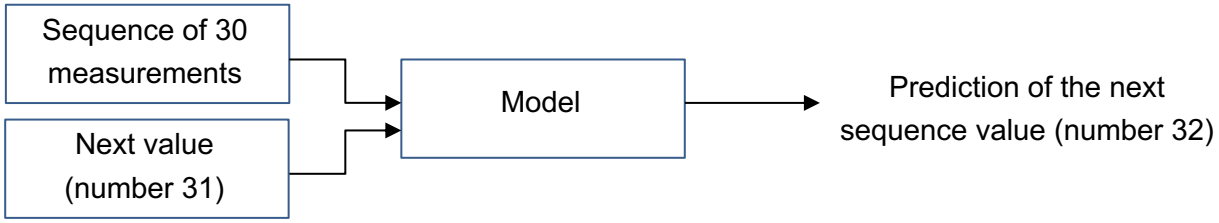


Figure 18: Schematic description of the prediction process

[44] https://yanlinc.medium.com/how-to-do-feature-selection-dimension-reduction-883c844aaaf6

The model is a standard implementation of an encoder-decoder transformer architecture using the PyTorch library[45] [46] [47] [48], with a self-implementation of the mask and the positional encoding.

```
[35] class Transformer(nn.Module):
        def __init__(self, feature_size, output_size, num_encoder_layers, num_heads, num_decoder_layers, device, dim_feedforward: int=2048, dropout: float =0.1, batch_first: bool = False):
            super(Transformer, self).__init__()

            encoder_layer = nn.TransformerEncoderLayer(d_model= feature_size, nhead= num_heads, dim_feedforward=dim_feedforward, dropout=dropout, device=device, batch_first=batch_first)
            decoder_layer = nn.TransformerDecoderLayer(d_model= feature_size, nhead= num_heads, dim_feedforward=dim_feedforward, dropout=dropout, device=device, batch_first=batch_first)

            self.encoder = nn.TransformerEncoder(encoder_layer, num_layers= num_encoder_layers)
            self.decoder = nn.TransformerDecoder(decoder_layer, num_layers= num_decoder_layers)

            self.output_size = output_size
            self.device = device

        def generate_square_mask(self, dim):
            return torch.triu(torch.ones(dim, dim) * float('-inf'), diagonal=1).to(self.device)

        def positional_encoding(seq_len: int, dim_model: int, device):

            position_encoding = torch.zeros(seq_len, dim_model)
            for pos in range(seq_len):
              for i in range(0, int(dim_model / 2)):
                 position_encoding[pos, 2 * i] = math.sin(pos / (10000 ** ((2 * i)/dim_model)))
                 position_encoding[pos, (2 * i) + 1] = math.cos(pos / (10000 ** ((2 * i)/dim_model)))

            return position_encoding.to(device)

        def forward (self, enc_input, dec_input):

            mask = self.generate_square_mask(len(enc_input))

            src_pos_enc = enc_input + self.positional_encoding(seq_len= enc_input.shape[1], dim_model= enc_input.shape[2], device= self.device)
            src_pos_dec = dec_input + self.positional_encoding(seq_len= dec_input.shape[1], dim_model= dec_input.shape[2], device= self.device)

            output = self.encoder (src=src_pos_enc, mask=None)
            output = self.decoder (tgt= src_pos_dec, memory= output, tgt_mask=mask, memory_mask=None)

            return output
```

3.2.1 Training and testing

The whole training process was implemented in a Jupyter Notebook using a pro subscription to Google Colab[49] with a Tesla P100-PCIE-16GB as GPU.
The model doesn't accept the raw data set in pandas format, therefore before the model will be fed with the data, the data needs to be prepared.

3.2.1.1 Preparing the data for training

I used Pytorch data loaders[50] to feed the models.
First, I had to make tuples of inputs (encoder and decoder) and their corresponding targets. I choose a sequence length of 30. Therefore, I choose sequences of 30 consecutive samples as encoder input, the next sample in the sequence (sample number 31) as decoder input, and the next sample (sample number 32) as target.

---

[45] https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoder.html
[46] https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoderLayer.html
[47] https://pytorch.org/docs/stable/generated/torch.nn.TransformerDecoder.html
[48] https://pytorch.org/docs/stable/generated/torch.nn.TransformerDecoderLayer.html
[49] https://colab.research.google.com
[50] https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

For example, if we have these 30 consecutive samples as encoder input:

	time	PLN1	PLN2	PLN3	ULL1	ULL2	ULL3	COS_PHI1	COS_PHI2	COS_PHI3	FREQ
0	2020-06-01 00:00:00	1141.0819	519.5834	482.9381	398.8613	400.1982	395.6010	0.8091	0.6864	0.4875	49.9927
1	2020-06-01 00:01:00	1145.1162	519.1807	491.4436	398.6934	400.1579	395.5431	0.8080	0.6903	0.4904	49.9779
2	2020-06-01 00:02:00	1140.9558	743.3837	484.9942	398.4367	400.1205	395.5259	0.8113	0.9274	0.4806	49.9782
3	2020-06-01 00:03:00	1151.9409	741.4836	487.4224	398.9800	400.4375	395.8621	0.8249	0.9123	0.4778	49.9850
4	2020-06-01 00:04:00	1142.1594	741.9858	486.7629	398.7133	400.3145	395.6446	0.8081	0.9291	0.4552	49.9856
5	2020-06-01 00:05:00	1151.4448	740.8891	483.2065	398.2193	399.6826	395.0238	0.8231	0.9152	0.4860	49.9879
6	2020-06-01 00:06:00	1140.3663	738.0733	484.5620	398.3398	400.0419	395.4245	0.8131	0.9238	0.5036	49.9863
7	2020-06-01 00:07:00	1141.8704	738.4373	488.9699	397.9988	399.6658	395.1119	0.8190	0.9177	0.4920	49.9829
8	2020-06-01 00:08:00	1145.1082	735.9846	491.7678	398.2296	399.7207	395.1026	0.8189	0.9272	0.4798	49.9942
9	2020-06-01 00:09:00	1143.8096	736.0773	488.6075	398.3539	400.0621	395.5134	0.8105	0.9192	0.4653	49.9910
10	2020-06-01 00:10:00	1149.9846	736.6874	490.1449	398.7785	400.2785	395.6080	0.8179	0.9236	0.4735	49.9860
11	2020-06-01 00:11:00	1141.6466	736.5790	534.6445	398.8537	400.6919	395.9734	0.8212	0.9196	0.5255	49.9889
12	2020-06-01 00:12:00	1146.1782	737.5667	533.0903	398.9976	400.7503	396.1033	0.8171	0.9237	0.5344	50.0066
13	2020-06-01 00:13:00	1142.1829	734.8600	528.9662	398.3488	399.8553	395.2602	0.8147	0.9146	0.5147	50.0162
14	2020-06-01 00:14:00	1141.0703	1027.0469	533.4843	398.4967	400.2210	395.6230	0.8102	0.9436	0.5354	50.0033
15	2020-06-01 00:15:00	1151.8193	734.8393	537.1331	398.2716	399.6821	395.0519	0.8109	0.9332	0.5239	50.0112
16	2020-06-01 00:16:00	1139.4744	1425.7758	532.0292	398.7186	400.3971	395.8348	0.8136	0.9735	0.5140	50.0059
17	2020-06-01 00:17:00	1147.1619	732.5831	535.5194	398.7247	400.3594	395.7264	0.8127	0.9253	0.5086	50.0188
18	2020-06-01 00:18:00	1141.3447	735.9214	531.3181	398.6631	400.5473	395.8600	0.8179	0.9202	0.5237	49.9924
19	2020-06-01 00:19:00	1146.7936	1375.8605	533.7009	398.6527	400.3847	395.8478	0.8122	0.9687	0.5274	50.0045
20	2020-06-01 00:20:00	1138.9264	739.8730	531.0894	398.5336	400.3839	395.6385	0.8091	0.9235	0.5042	50.0041
21	2020-06-01 00:21:00	1138.5939	738.0576	531.9845	398.5486	400.4684	395.8480	0.8238	0.9242	0.5327	49.9999
22	2020-06-01 00:22:00	1146.8696	1318.0564	540.4203	398.9987	400.6716	396.1212	0.8155	0.9675	0.5314	50.0046
23	2020-06-01 00:23:00	1141.7528	740.3690	531.5391	399.2563	401.2218	396.5398	0.8191	0.9182	0.5222	50.0112
24	2020-06-01 00:24:00	1146.1506	1600.3192	537.8782	399.2487	400.8439	396.3313	0.8214	0.9753	0.5138	50.0189
25	2020-06-01 00:25:00	1144.5024	736.5547	531.6068	398.9716	400.6333	396.0279	0.8098	0.9179	0.5218	50.0072
26	2020-06-01 00:26:00	1141.8391	961.4946	532.5130	399.1673	401.0371	396.4368	0.8105	0.9523	0.5250	50.0169
27	2020-06-01 00:27:00	1144.8197	1415.6816	527.9114	399.3638	400.9925	396.4250	0.8121	0.9742	0.5105	50.0144
28	2020-06-01 00:28:00	1144.8068	738.8981	534.8634	399.3053	401.1208	396.5471	0.8196	0.9283	0.5416	50.0128
29	2020-06-01 00:29:00	1146.2703	735.5371	536.1163	399.3404	400.9826	396.3327	0.8206	0.9138	0.5125	50.0144

We should have this sample as decoder input:

30	2020-06-01 00:30:00	2443.9519	1476.1388	528.6713	398.6752	400.3599	395.7498	0.9835	0.9761	0.5250	49.9861

And this sample as target for the training:

31	2020-06-01 00:31:00	2397.3098	737.0045	534.5628	398.6799	400.7572	396.0622	0.9832	0.9175	0.5211	49.9940

The strategy taken to have the maximum number of pairs input-target is to start with 0 as the starting point. Take the next 29 together with 0 as input. Then the next as decoder input and the following one as target. And start the process again, but with 1 as the starting point. Following this strategy, we obtain 63300 pairs of sequences with a length of 30 samples.

```
[26] def create_sequece_dataloaders_new(dataset_norm):

       # Create a dataset with pairs data / next /Target (in this case data is one
       # sequence of 30 measures (18 features), next is the next value in the sequence
       # and target is the following value with the
       # measurements (18 features)). When you plug in one measurement, the model should out the next measurement

       pair_set = []

       for i in tqdm(range(len(dataset_norm) - 60)):
         data = np.array(dataset_norm.iloc[i:i+30, 1:])
         next = np.array(dataset_norm.iloc[i+30, 1:], dtype= float)
         target = np.array(dataset_norm.iloc[i+31, 1:], dtype= float)

         pair_set.append((data, next, target))

       dataset_pairs = np.array(pair_set)
```

The next step in preparing the data for training is to split it into 2 datasets. One for training the model and one for testing the results of the training. For this purpose, I used the library train_test_split from sklearn[51]. With a test_size of 0.1 and shuffle=true to achieve randomness in the splitting

```
training_data_pairs, testing_data_pairs = train_test_split(dataset_pairs, test_size=0.1)

data = []
next = []
target = []

for i in training_data_pairs:
  data.append(i[0])
  next.append(i[1])
  target.append(i[2])

training_data = torch.from_numpy(np.array(data)).float().to(device)
training_next = torch.from_numpy(np.array(next)).float().to(device)
training_target = torch.from_numpy(np.array(target)).float().to(device)

data = []
next = []
target = []

for i in testing_data_pairs:
  data.append(i[0])
  next.append(i[1])
  target.append(i[2])

test_data = torch.from_numpy(np.array(data)).float().to(device)
test_next = torch.from_numpy(np.array(next)).float().to(device)
test_target = torch.from_numpy(np.array(target)).float().to(device)
```

After this process I had a training data set with 56970 sequences of 30 samples for training and a test data set with 6330 sequences of 30 samples.

The last step is to create the data loaders. I created two data loaders with minibatches of 30 sequences. One for training purposes and one for testing the data. Both with the data shuffled. That ends with 950 batches of 60 sequences for the train data set and 106 batches of 60 sequences for the test data set.

```
# Create data loader to feed the model in mini batches
loader_train = torch.utils.data.DataLoader(
    dataset=torch.utils.data.TensorDataset(training_data, training_next, training_target),
    batch_size=60,
    shuffle=True
)

# Create data loader for testing the model
loader_test = torch.utils.data.DataLoader(
    dataset=torch.utils.data.TensorDataset(test_data, test_next, test_target),
    batch_size=60,
    shuffle=True
)

return loader_train, loader_test
```

[51] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

At this moment, I had the data prepared to use it to feed the model and ready to start the training.

3.2.1.2 Baseline model

The first question that arose when training any model is answering the question: How do we know if our predictions are good enough?
To answer this question, we need a baseline model. A baseline model is a simple, but robust, way to make predictions in our input database[52]. The results of our final model should be, at least, better than the result of our baseline model. That would be a good starting point to continue developing our model.
There are different strategies to create a good baseline model. In regression tasks (like the task in this thesis), the most common are:

- Predict the mean of the input data
- Predict the input as output

In the first baseline model I used, I used the prediction of input as output.

```
criterion = nn.MSELoss()

losses_train = []

for i in loader_train:
  output = i[0]
  target = i[1]
  loss = criterion(output, target)
  losses_train.append(loss.item())

losses_test = []

for i in loader_test:
  output = i[0]
  target = i[1]
  loss = criterion(output, target)
  losses_test.append(loss.item())

print("Training set")
print("Mean Loss of baselinemodel: ", np.mean(losses_train))
print("Standard deviation Loss of baselinemodel: ", np.std(losses_train))
print('\n')
print("Test set")
print("Mean Loss of baselinemodel: ", np.mean(losses_test))
print("Standard deviation Loss of baselinemodel: ", np.std(losses_test))
print('\n')
```

[52] https://machinelearningmastery.com/how-to-know-if-your-machine-learning-model-has-good-performance/

Using the input as output and comparing it with the ground truth and using the mean square error[53] (MSE) as a loss function,

$$L(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n}(y_i - \hat{y}_i)^2$$

I obtained the following results:

- The mean of the loss through the whole training dataset is: 0.472
  With a standard deviation of 0.091
- The mean of the loss through the whole test dataset is: 0.4825
  With a standard deviation of 0.087

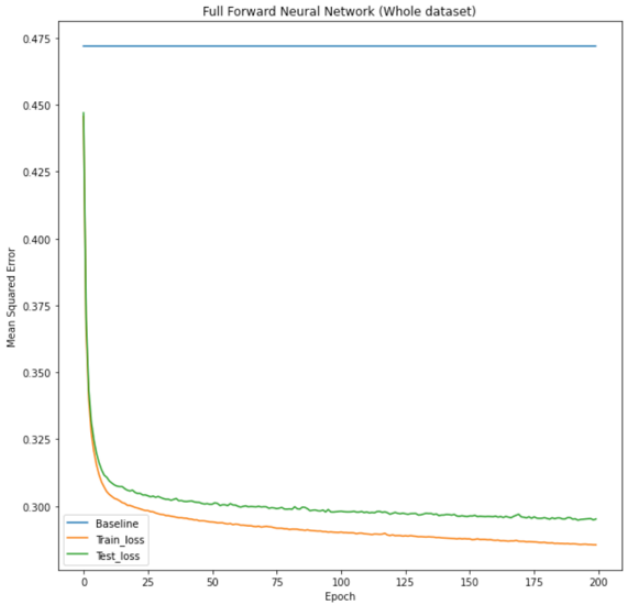
I also created a second baseline model, using a normal FFN to train sequences of data.
The idea is to train a FFN that output one singular sample and compares it with the next sample in the data set. I am aware that I am not considering the samples as a series but only as isolated samples, but I am only using this approach to make a better baseline model.
The model I used was a simple FFN in PyTorch[54]. With 4 hidden layers, input of 18 dimensions and output of 18 dimensions, with ReLu[55] as an activation function.

```
print(model_FFN_whole)

ANN_relu(
  (linear1): Linear(in_features=18, out_features=180, bias=True)
  (linear2): Linear(in_features=180, out_features=360, bias=True)
  (linear3): Linear(in_features=360, out_features=360, bias=True)
  (linear4): Linear(in_features=360, out_features=180, bias=True)
  (linear5): Linear(in_features=180, out_features=18, bias=True)
)
```

Trained over 200 epochs with MSE[56] as loss function and Stochastic Gradient Descend as optimizer[57] method.
After splitting the data into train and test samples in a ratio of 0.1. I obtained following results after the training:

[53] https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-squared-error
[54] https://pytorch.org/docs/stable/generated/torch.nn.Linear.html
[55] https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html
[56] https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html
[57] https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

- Mean loss in training data: 0.286
- Mean loss in test data: 0.295
- Training time: approximately 6,5 minutes
- Training time per epoch: approximately 2 seconds



What indeed is an improving from the first baseline model.

These results are better than the first baseline model, and our final model should improve them too. We must consider that this baseline model is not based on a well-trained researched FFN. With more time and effort, I am sure that I could improve these baseline results.

3.2.1.3 First training of the transformer model

Schematically, one training iteration looks like this:

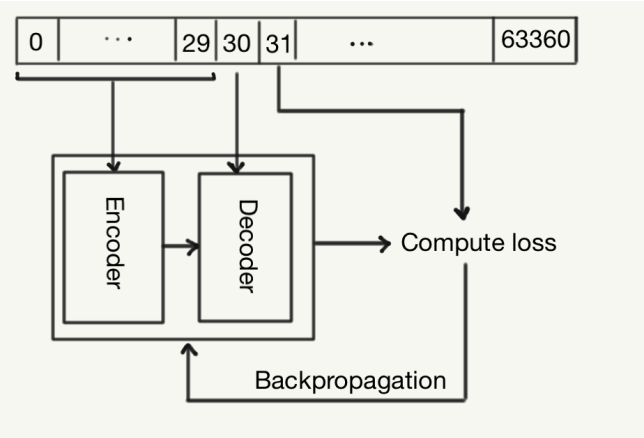


Figure 19: Schematic description of the training process

First, we need to create an instance of the model. In the first step of the training process, I choose standard hyperparameters:

- 6 Encoder Layers
- 1 Decoder Layer
- 6 Attention heads
- An internal FFN of 2048 units in the hidden layer

```
# Initialize Transformer Model and Optimizer

model_transformer = Transformer (num_encoder_layers=6,
                      num_decoder_layers=1,
                      feature_size=18,
                      output_size=18,
                      num_heads=6,
                      dim_feedforward=2048,
                      device = device,
                      batch_first=False)
```

As an optimizer, I choose the PyTorch implementation of Stochastic Gradient Descend[58], with a learn rate of 0.01 and as loss function criterion, the PyTorch implementation of Mean Square Error[59].

I trained the model over the data loaders computed before and over 200 epochs.

In every epoch the model is trained over the whole training data set. Compute the mean loss over the training data set and compute the mean loss over the whole test data set. Both

[58] https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

[59] https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html

values are stored to track the performance of the training in every epoch. Recall, both train and test data sets were computed in the previous sub-chapter (3.2.1.1.).
One epoch of training works as follow:

- Iterate over the whole train loader (Every batch in the loader is a tuple, containing the encoder input sequence, the decoder input element, and the ground truth element that corresponds to that input, named as target)
- Perform the forward-pass over one batch of data.
- Compute the MSE with the output of the previous step and the target.
- Perform backpropagation and gradient descend (the gradients are normalized[60])

```
for e in range(1, n_epochs + 1):

  print(f'Epoch: {e} of {n_epochs}')
  print('Training...')
  model.train()

  for i in tqdm(train_loader):

    input = i[0]
    out = i[1].unsqueeze(0).permute(1,0,2)
    target = i[2].unsqueeze(0).permute(1,0,2)

    net_out = model.forward(input, out)

    #Compute loss
    loss = criterion(net_out, target)

    optimizer.zero_grad()

    #Backpropagation
    loss.backward()

    torch.nn.utils.clip_grad_norm_(model.parameters(), 0.5)

    #Optimization
    optimizer.step()
```

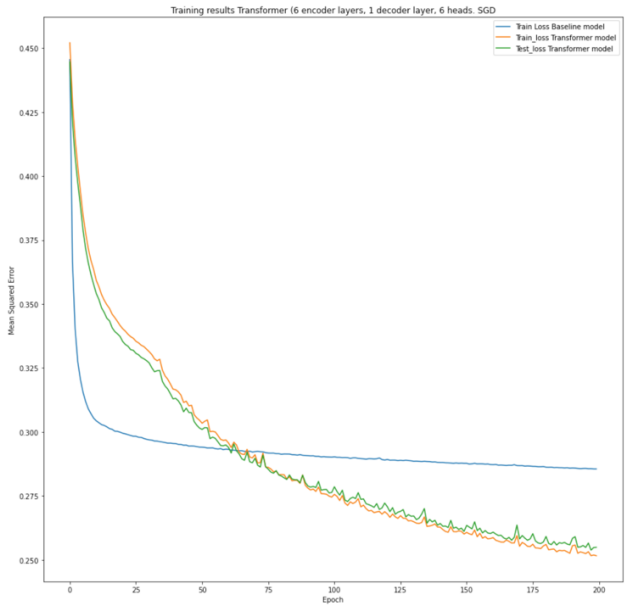
After the training, we compute the mean error over the whole training data set and over the whole test data set, performing forward pass, comparing with the target values, and storing the results.

[60] https://pytorch.org/docs/stable/generated/torch.nn.utils.clip_grad_norm_.html

```
print('\nTest with training set')
losses_train = []
model.eval()
with torch.no_grad():
  for i in tqdm(train_loader):

    input = i[0]
    out = i[1].unsqueeze(0).permute(1,0,2)
    target = i[2].unsqueeze(0).permute(1,0,2)

    net_out = model.forward(input, out)

    #Compute loss
    losses_train.append (float(criterion(net_out, target).item()))


print('\nCurrent Mean loss Train Set: ', np.mean(losses_train))
epoch_loss_train.append(losses_train)

print('\nTest with test set')
losses_test = []
model.eval()


with torch.no_grad():
  for i in tqdm(test_loader):

    input = i[0]
    out = i[1].unsqueeze(0).permute(1,0,2)
    target = i[2].unsqueeze(0).permute(1,0,2)

    net_out = model.forward(input, out)

    #Compute loss
    losses_test.append (float(criterion(net_out, target).item()))

print('\nCurrent Mean loss Test Set: ', np.mean(losses_test))
epoch_loss_test.append(losses_test)
```

And now we only must repeat the process the number of epochs that we desire, till the test loss achieved gives a satisfactory result. In this case, I trained the model over 200 epochs. Obtaining the following results and comparing them with the baseline model:

- Mean loss in training data: 0.2516
- Mean loss in test data: 0.2549
- Training (and testing) time: approximately 88 minutes
- Training (and testing) time per epoch: approximately 26 seconds



The results show an improvement in relation to both baseline models. And proving that use a transformer model is a valid proof of concept for this task. The training curve shows that more epochs could reduce even more the error rate. And that improvements in the model are possible.

3.2.2 Improving the model

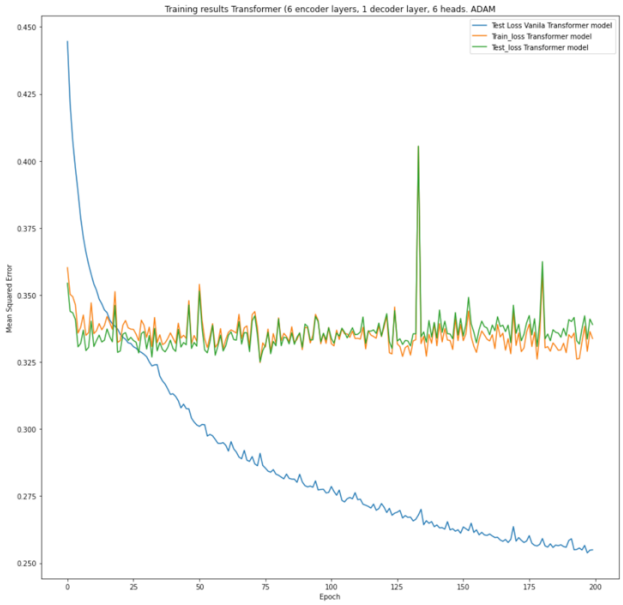
The first improvement I tried was to change the optimization algorithm. Instead of an SGD I used ADAM[61]. Most precisely the PyTorch implementation of ADAM[62] with a learn rate of 0.01.

Keeping without change the other hyperparameters and training over the same train and test data set and also over 200 epochs, I obtain the following results comparing them to the results obtained with the first transformer implementation:

- Mean loss in training data: 0.3338
- Mean loss in test data: 0.3390
- Training (and testing) time: approximately 105 minutes
- Training (and testing) time per epoch: approximately 32 seconds

[61] https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

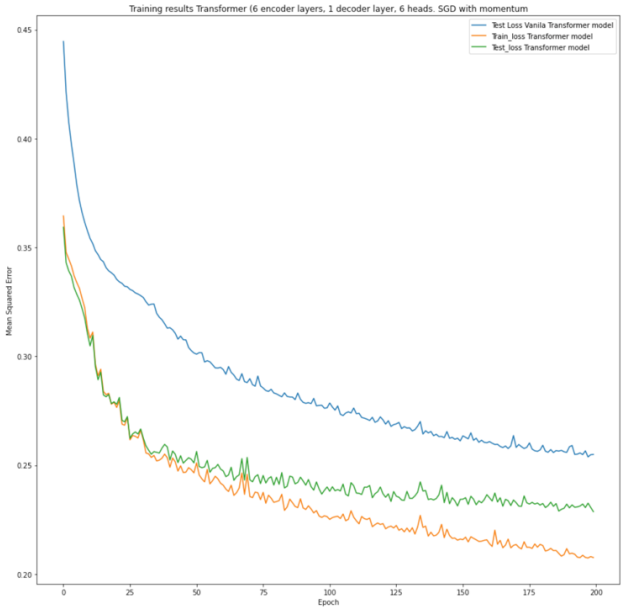
[62] https://pytorch.org/docs/stable/generated/torch.optim.Adam.html



This training's results didn't show any improvements than using SGD. Very unstable error rates both in train and test sets. In this case is not recommendable to use Adam as an optimization algorithm.

The second improvement I tried was to return to the SGD optimization algorithm but adding a momentum. I repeat the training with the same data, the same hyperparameters and with the PyTorch implementation of SGD, this time using a learning rate of 0.01 and a momentum of 0.9. I obtain the following results:

- Mean loss in training data: 0.2076
- Mean loss in test data: 0.2287
- Training (and testing) time: approximately 93 minutes
- Training (and testing) time per epoch: approximately 28 seconds

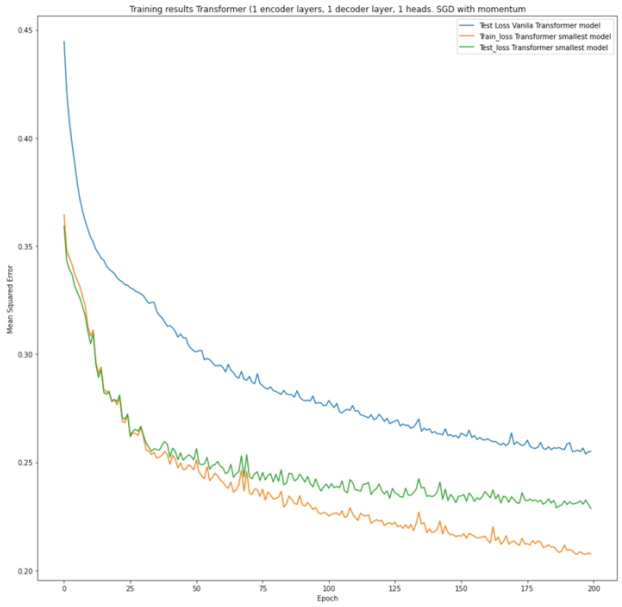


As expected, the results show an improvement when compared with the results of SGD without momentum. Significantly more stable than using ADAM and it doesn't show any signs of overfitting in the first 200 epochs. These results can be improved training the model through more epochs.
The training time is very similar as using SGD without momentum. These results show that, using SGD with momentum is a better choice in the learning process than ADAM or SGD without momentum.

Other possible improvements could be reached by changing the model hyperparameters. I changed the number of encoder layers, decoder layers and heads to see which changes could be an improvement to the model.

First, I build a small version of the model to see how the results change using the smallest model possible. I implement a model with only 1 encoder layer, 1 decoder layer, and 1 multi-head attention layer. I trained during 200 epochs using SGD with momentum as optimizer and I obtained the following results:

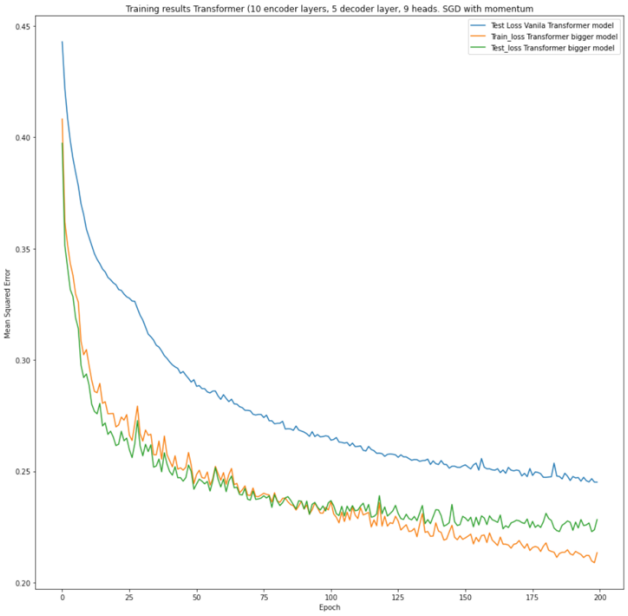
- Mean loss in training data: 0.2244
- Mean loss in test data: 0.2354
- Training (and testing) time: approximately 48 minutes
- Training (and testing) time per epoch: approximately 14 seconds



As expected, the error rate after 200 epochs is higher than using a bigger model, but using momentum in the optimizer give better results than a bigger model without using momentum. This prove that use SGD with momentum is a better choice than the other optimizers tried. As shown in the results, a smaller model implies better time performance. More research could show what will be the better choice in the tradeoff between time performance and error results.

As final research, I implemented a bigger model to see if there is a significant improve in the error results and to compare them with the smaller ones, in terms both of time and error performance. I implemented a model with 10 encoder layer, 5 decoder layer, and 9 multi-head attention layers. I trained during 200 epochs using SGD with momentum as optimizer and I obtained the following results:

- Mean loss in training data: 0.2134
- Mean loss in test data: 0.2283
- Training (and testing) time: approximately 231 minutes
- Training (and testing) time per epoch: approximately 69 seconds



Although the results are better than the smallest model results, they are not better than the standard model with SDG with momentum. This prove that more research in changing the hyperparameters could bring better results, but a biger model doesn't imply always better results . Moreover, around epoch 150, the model is starting to be unstable in the results with the test set, that could be an early sing of overfitting. As shown in the results, a bigger model implies worst time performance. As in the case with smaller model, further research could show what will be the best choice in the tradeoff between time performance and error results.

4 Conclusions

Recalling the idea behind this project: To show that a transformer architecture model, when a sequence of 30 samples is provided, could predict the next sample. Then, this new sample could be compared with the real sample provided by the IOT sensors. In other words, to compare what should be in a correct function of the machines (output of the model) with what is (real data form sensors). If this difference is bigger than a fixed tolerance, that could show that there will be a malfunction soon.

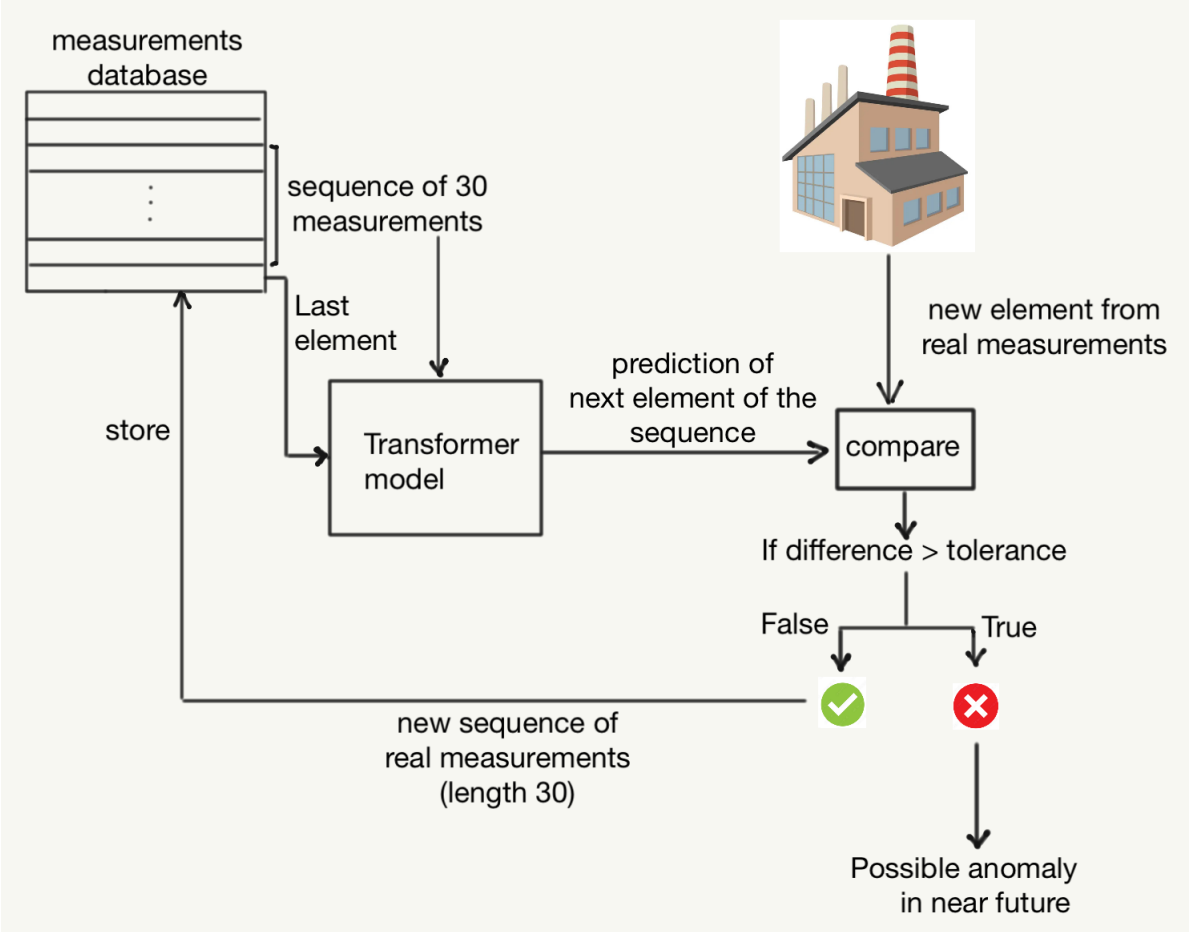


Figure 20: Schematic description of the entire project

After the first training with a kind of "vanilla"[63] model, the results in form of Mean Squared Error over all samples and features of the sequence show a big improvement over the

[63] https://en.wikipedia.org/wiki/Vanilla_software

baseline models. Further improvements of this model showed that this error could be minimized even more, as we can see in this table:

Algorithm parameters	MSE Training set	MSE Test set	Training Time (200 Epochs)	Training time per Epoch	Forward-pass Time per Sample
Baseline model, FFN, 4 hidden layers 200 epochs	0.2862	0.2951	7 min.	2 sec	0.1 ms
6 E. Layers, 1 D. Layer, 6 Heads. SGD. 200 Epochs	0.2516	0.2549	88 min.	26 sec.	32 ms
6 E. Layers, 1 D. Layer, 6 Heads. ADAM. 200 Epochs	0.3338	0.3390	105 min.	32 sec.	32 ms
6 E. Layers, 1 D. Layer, 6 Heads. SGD with momentum. 200 Epochs	0.2076	0.2287	93 min.	28 sec.	32 ms
1 E. Layers, 1 D. Layer, 1 Heads. SGD with momentum. 200 Epochs	0.2244	0.2354	48 min.	14 sec.	24 ms
10 E. Layers, 5 D. Layers, 9 Heads, SGD with momentum. 200 Epochs	0.2134	0.2283	231 min	69 sec	59 ms

In other aspect, I think the fact that there are very strong correlated features and the fact that some of the features have a distribution similar to a normal distribution could open a possibility to improve the model working on the data. Therefore, improvements in the future could be made, for them we could take the previous computed models as a new baseline model.

Recalling the initial assumption: to be able predict which values should be receive from the sensors to attest a correct functioning of the machines in the factory. At this moment I cannot prove the validity of it. For that I would need more data and more experiments in the real case. For example, I don’t know at this moment when these errors, though being a big improvement from the baseline models, would be acceptable, or I don’t know at this moment how big, and in which features, the difference between predicted and real values should be to attest a malfunction in the machines of the factory.

In theory, and looking at the scale of these errors, I think that this thesis shows a good proof of concept starting point for further research.

Bibliography

[1] G. Zerveas, S. Jayaraman, D. Patel, A. Bhamidipaty, and C. Eickhoff, ‘A Transformer-based Framework for Multivariate Time Series Representation Learning’, in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, Virtual Event Singapore, Aug. 2021, pp. 2114–2124. [Online]. Available: https://dl.acm.org/doi/10.1145/3447548.3467401

[2] M. Zaheer *et al.*, ‘Big Bird: Transformers for Longer Sequences’, *arXiv:2007.14062 [cs, stat]*, Jan. 2021, Accessed: Jan. 30, 2022. [Online]. Available: http://arxiv.org/abs/2007.14062

[3] A. Vaswani *et al.*, ‘Attention Is All You Need’, *arXiv:1706.03762 [cs]*, Dec. 2017, Accessed: Jan. 30, 2022. [Online]. Available: http://arxiv.org/abs/1706.03762

[4] N. Wu, B. Green, X. Ben, and S. O’Banion, ‘Deep Transformer Models for Time Series Forecasting: The Influenza Prevalence Case’, *arXiv:2001.08317 [cs, stat]*, Jan. 2020, Accessed: Jan. 30, 2022. [Online]. Available: http://arxiv.org/abs/2001.08317

[5] Samuel Lynn-Evans, ‘How to code The Transformer in Pytorch’, Sep. 27, 2018. https://towardsdatascience.com/how-to-code-the-transformer-in-pytorch-24db27c8f9ec#1b3f

[6] J. Baek, ‘A Simple Example of Causal Attention Masking in Transformer Decoder’, Sep. 06, 2021. https://medium.com/@jinoo/a-simple-example-of-attention-masking-in-transformer-decoder-a6c66757bc7d

[7] M. Phy, ‘Illustrated Guide to Transformers- Step by Step Explanation’, May 01, 2020. https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f74876522bc0

[8] S. Kierszbaum, ‘Masking in Transformers’ self-attention mechanism’, Jan. 27, 2020. https://medium.com/analytics-vidhya/masking-in-transformers-self-attention-mechanism-bad3c9ec235c

[9] A. Kazemnejad, ‘Transformer Architecture: The Positional Encoding’, Sep. 9, 2019. https://kazemnejad.com/blog/transformer_architecture_positional_encoding

[10] J. Hochreiter, ‘Untersuchungen zu dynamischen neuronalen Netzen‘, Jun. 15, 1991. [Online]. Available: https://people.idsia.ch/~juergen/SeppHochreiter1991ThesisAdvisorSchmidhuber.pdf

[11] S. Hochreiter and J. Schmidhuber, ‘Long Short-Term Memory’, *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10.1162/neco.1997.9.8.1735. [Online]. Available: https://direct.mit.edu/neco/article/9/8/1735-1780/6109

[12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, ‘Learning representations by back-propagating errors’, *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986, doi: 10.1038/323533a0. [Online]. Available: http://www.nature.com/articles/323533a0

[13] K. Hornik, M. Stinchcombe, and H. White, ‘Multilayer feedforward networks are universal approximators’, *Neural Networks*, vol. 2, no. 5, pp. 359–366, Jan. 1989, doi: 10.1016/0893-6080(89)90020-8. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0893608089900208

[14] C. Santana Vega, *¿Por qué estas REDES NEURONALES son tan POTENTES? TRANSFORMERS Parte 2 (YouTube)*, Sep. 14, 2021. [Online]. Available: https://www.youtube.com/watch?v=xi94v_jl26U

[15] C. Santana Vega, Las REDES NEURONALES ahora prestan ATENCIÓN! TRANSFORMERS ¿Cómo funcionan? (YouTube), Sep. 27, 2021. [Online]. Available: https://www.youtube.com/watch?v=aL-EmKuB078&t=839s

[16] ‘torch.nn.Module’. https://pytorch.org/docs/stable/generated/torch.nn.Module.html

[17] ‘torch.nn.TransformerEncoder’. https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoder.html

[18] ‘torch.nn.Transformer’. https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html

[19] ‘torch.nn.TransformerDecoder’. https://pytorch.org/docs/stable/generated/torch.nn.TransformerDecoder.html

[20] ‘torch.nn.Linear’. https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

[21] https://www.simplypsychology.org/short-term-memory.html

[22] https://dev.to/shambhavicodes/let-s-pay-some-attention-33d0

[23] https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[24] https://pytorch.org/docs/stable/generated/torch.optim.Adam.html

[25] https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

[26] https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html

[27] https://pytorch.org/docs/stable/generated/torch.nn.utils.clip_grad_norm_.html

[28] https://machinelearningmastery.com/how-to-know-if-your-machine-learning-model-has-good-performance/

[29] https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-squared-error

[30] https://yanlinc.medium.com/how-to-do-feature-selection-dimension-reduction-883c844aaaf6

List of abbreviations

IOT	Internet Of Things (https://en.wikipedia.org/wiki/Internet_of_things)
.csv	Comma Separated Value (https://en.wikipedia.org/wiki/Comma-separated_values)
NaN	Not a Number (https://en.wikipedia.org/wiki/NaN)
RNN	Recurrent Neural Network (https://en.wikipedia.org/wiki/Recurrent_neural_network#cite_note-9)
FFN	Feed Forward Network (https://en.wikipedia.org/wiki/Feedforward_neural_network)
PCA	Principal Component Analysis (https://en.wikipedia.org/wiki/Principal_component_analysis)
t-SNE	t-distributed stochastic neighbor embedding (https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding)
NLP	Natural Language Processing (https://en.wikipedia.org/wiki/Natural_language_processing)
MSE	Mean Squared Error (https://en.wikipedia.org/wiki/Mean_squared_error)