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 TRANSFORMER 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, Mai 18, 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 dataset, and will show a model that, using transformers over a sequence of multidimensional tabular data, can forecast which should be the next element of that sequence. Showing indeed that a transformer architecture is suitable dealing with time series tasks.

**Keywords:** Computer science, artificial intelligence, machine learning, data science, transformers, time series, tabular data.

# Zusammenfassung

Im Dezember 2017 wurde die Transformer-Architektur als vielversprechendste Lösung für Aufgaben der Verarbeitung natürlicher Sprache (NLP) vorgestellt und ist seitdem Stand der Technik für diese Aufgaben. Bei NLP-Aufgaben werden Wörter jedoch als Vektoren in einem mehrdimensionalen Raum kodiert, und wenn Sätze eine Folge von Wörtern sind, sollte theoretisch jede Aufgabe, die sich mit Folgen von mehrdimensionalen Vektoren befasst, für eine Lösung mit einem Transformer Modell geeignet sein.
In dieser Arbeit wird der Autor das Problem, das die Forschung des Transformators inspiriert hat, und die Details seiner Architektur auf einfache Weise erklären, eine Datenanalyse in einem bestimmten Datensatz durchführen und ein Modell zeigen, das mit Hilfe von Transformers über eine Sequenz von mehrdimensionalen Tabellendaten vorhersagen kann, welches das nächste Element dieser Sequenz sein sollte. Es wird gezeigt, dass die Transformer-Architektur geeignet ist, Zeitreihenaufgaben zu bewältigen.

**Schlüsselwörter:** Informatik, künstliche Intelligenz, maschinelles Lernen, Datenwissenschaft, Transformers, Zeitreihen, tabellarische Daten.

# Note of thanks

Bettina Horvath for help with my English and basically for everything. Jesus Seijás for his advice and corrections. My supervisor Dietmar Millinger for his lessons, advice, and work. And I would like to thank all my family and friends for all the support over the years.

# Summary

# 1 Introduction

Since its advent in December 2017, the transformer model revolutionized the world of machine learning. Soon after the first implementation they became state of the art for Natural Language Processing Tasks. But they can be used for many different use cases. The main subject of this Thesis is to state a proof of concept that shows that Transformers can be used for a time series task using tabular data.

I will divide this thesis in two main parts:

In the first part I will explain in detail how is the architecture of a transformer model, how it works and what were the motivation behind its develop. I think as a future machine learning engineer is important to understand how a model works internally. Only with this understanding it can be used with full knowledge and not only as a black box. In other words, to understand and explain concepts like attention mechanism, positional encoding or the scaled dot product attention, the internal mechanism that makes possible the encoder or the decoder, or how a transformer model can make predictions about the next element of a sequence. This is my main motivation for the first part of this thesis.

In the second I will present a use case using time series with tabular data. I will describe the process of data analysis, the process of coding and tuning of the transformer model and the results given after the training. The use case is based in electrical data from a German factory. 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. Therefore, if different 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 or an anomaly.

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. The purpose of this thesis is to try, analyse and prove if a transformer model could be a good choice to deal with time series data.

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.[1]

[1] https://github.com/SergioTallo/Bsc_Thesis

# 2 Transformer 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[2]. But their capacity to work with ordered sequences of data is limited, 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) [3], in which a recurrent layer is added to a traditional FFN to deal with the previous samples in the training, testing and results process.

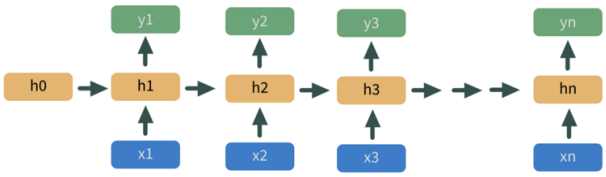



Figure 1: Schematic description of a recurrent neural network[2]

[2] Kurt Hornik, Maxwell Stinchcombe, Halbert White, 'Multilayer feedforward networks are universal approximators',Neural Networks,Volume 2, Issue 5,1989,Pages 359-366,

[3] 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[4], 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.

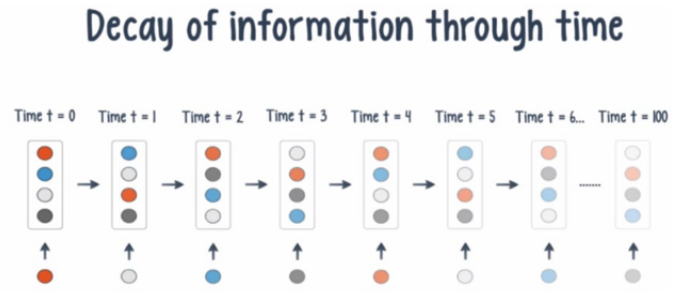



Figure 2: Schematic description of the decay of information[5]

### 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[6] 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 as follows[7]:

---

[4] https://www.simplypsychology.org/short-term-memory.html
[5] https://dev.to/shambhavicodes/let-s-pay-some-attention-33d0
[6] Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (1986a). "Learning representations by back-propagating errors". Nature. 323 (6088): 533–536.
[7] Mathematical computations based on concepts learned in the “Machine Learning Supervised techniques” subject in the JKU Linz by Prof. Johannes Kofler

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

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

$\delta_k$ : 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:

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

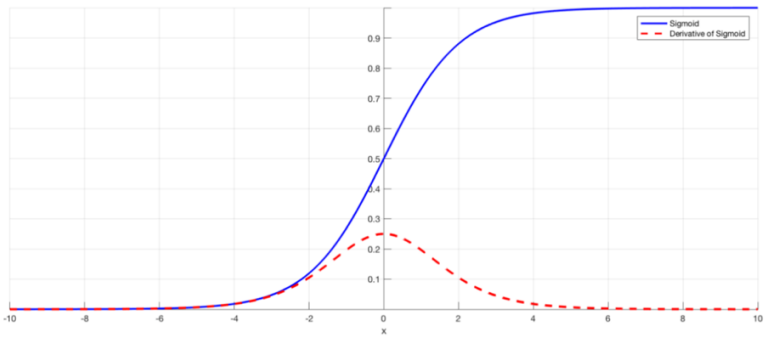



Figure 3: Graph of the sigmoid function and its derivative[8]

[8] 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:

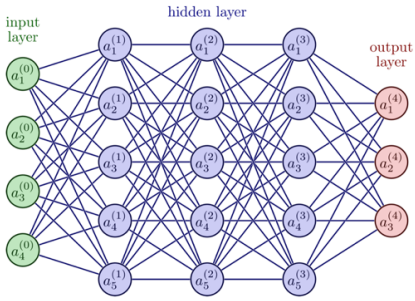



Figure 4: Schema of a small deep neural network[9]

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):

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

[9] 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

$$\delta^{(3)} = 0.25 \sum_{i=1}^{3} \delta^{(4)} w_{1^{(3)} i^{(4)}}$$

- 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 $\delta$-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[10] function:

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

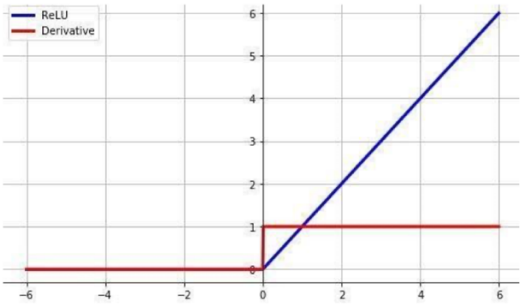



Figure 5: Graph of the ReLU function and its derivative[11]

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[12]. 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.

---

[10] Fukushima, K. (1969). "Visual feature extraction by a multilayered network of analog threshold elements". IEEE Transactions on Systems Science and Cybernetics.

[11] Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks, Tomasz Szandała. https://arxiv.org/pdf/2010.09458.pdf

[12] 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.

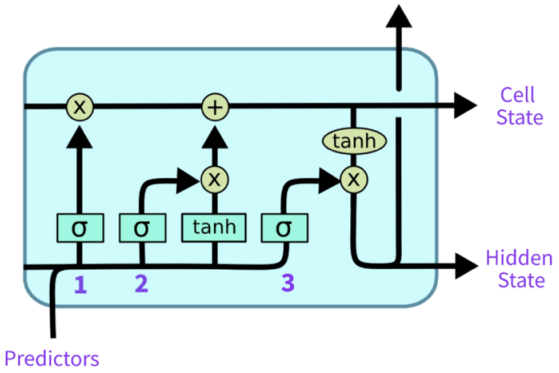



Figure 6: Schematic description of a LSTM unit[13]

Those approaches do not solve the vanishing gradient problem entirely. The first 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. In order to start one of the sequence samples, the sample before it must be finished.

## 2.2 Attention is all you need. Transformer 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[14].

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

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

Using this concept of attention, in June of 2017 (the last revision of the paper was in December 2017), a Team at Google Brain[15] presented a paper explaining a new machine learning model proposing that, attention itself is enough to learn the context of a series-based dataset called “Attention is all you need”[16], 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.

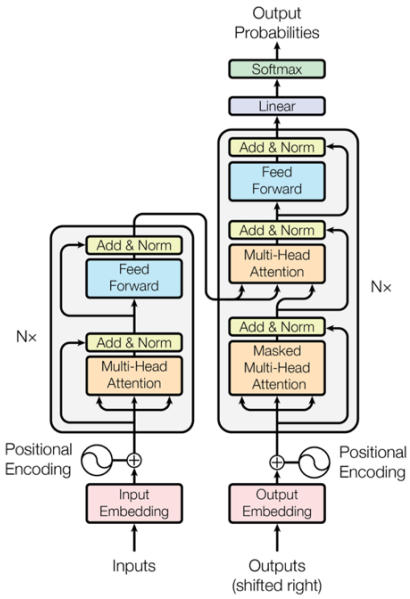



Figure 7: Schematic description of the transformer architecture[13]

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 dataset 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

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

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

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 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 or which after, 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:

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

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$[17]

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$[16]

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

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

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

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.

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

```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
```

Code snippet 1: Transformer positional encoder

## 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.

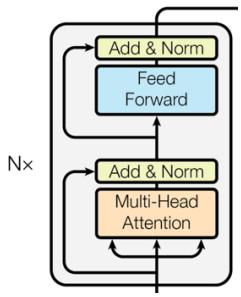



Figure 8: Schematic description of the encoder layer[18]

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.

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

### 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. 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.

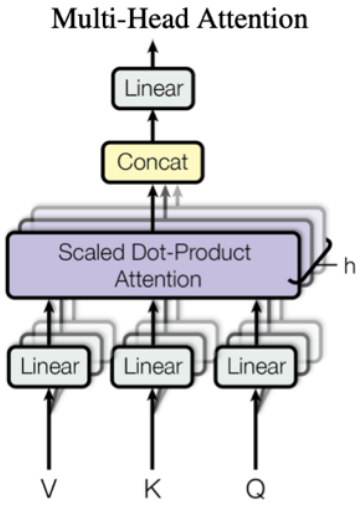



Figure 9: Schematic description of the multi-head attention[19]

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):

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

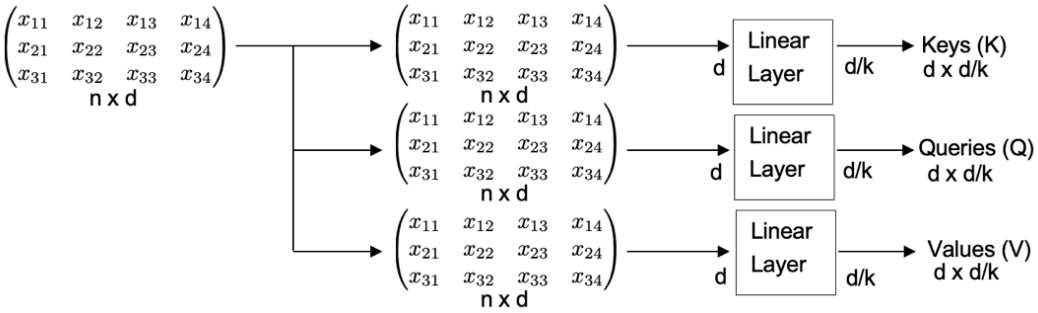
| | The | boy | Is | playing | with | his | brother |
|---|---|---|---|---|---|---|---|
| The | 1.00 | 0.80 | 0.01 | 0.02 | 0.05 | 0.04 | 0.03 |
| boy | 0.80 | 1.00 | 0.90 | 0.75 | 0.70 | 0.90 | 0.80 |
| is | 0.01 | 0.90 | 1.00 | 0.80 | 0.03 | 0.02 | 0.01 |
| playing | 0.02 | 0.75 | 0.80 | 1.00 | 0.50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

Figure 10: Example of Attention matrix

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” also has a big relationship with “is” (verb) and with “brother”. This matrix converts into numbers the relationships of the words in this 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[20]. These have the number of data features as the number of inputs and the number of features divided (floor division) by the number of heads (k) as the number of outputs. Three different matrices are calculated in these linear layers:

- Keys
- Queries
- Values




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

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

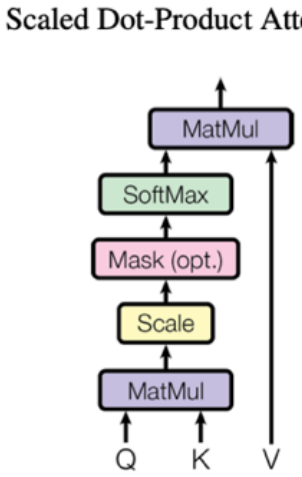



Figure 11: Schematic description of the Scaled Dot-Product Attention[21]

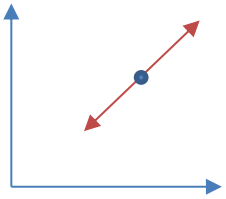
It works as follows:

For simplicity, let's assume 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 relationship of each sample in the sequence with every other sample in the same sequence (and to each other). To do this the output matrices Queries and Keys are used. Both come from the same input sequence; therefore, each sample is a transformation of the same original matrix. Therefore, this relationship is based on the similarity of each sample transformed in the Queries matrix and every sample transformed in the Keys matrix.
Since every sample is a one-dimensional vector, the mathematical explanation for this similarity is based on its direction. If two vectors are similar, their directions in a vector space are also similar. The maximum similarity is given if the vectors point in the same direction ($\vec{v} = \vec{w}$). Otherwise, the similarity is minimal if the vectors point in opposite directions ($\vec{v} = -\vec{w}$).

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

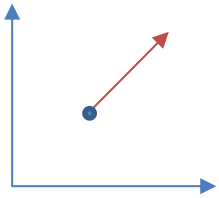
[21] https://arxiv.org/pdf/1706.03762.pdf

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 get the following square matrix with dimensions (sequence length x sequence length)

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

When calculating the similarity matrix, it can be important to mask the results[22]. This operation is optional, but important in many use cases. The encoder's masking task is to transform the attention values to 0 wherever there is only padding.
I will discuss the masking task itself when I talk about the decoder.
The next step is to pass the similarity matrix through a softmax function to transform the existing values into values between 1.00 for maximum similarity and 0.00 for minimum similarity.

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

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

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

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

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

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

```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
```

Code snippet 2: Scaled dot product attention

Currently, we assume only one self-attention head. If there were more than one head, the results (matrix with 2 dimensions, sequence length x (sequence length / number of heads)) would have to be concatenated to a resulting matrix (sequence length x sequence length). Assume 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 scaled dot product attention goes through another linear layer and the result is added to the original matrix (after the positional encoding) and the resulting matrix is normalized.

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

    # https://pytorch.org/docs/stable/generated/torch.nn.Module.html
    # https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

    def __init__(self, d_model: int, d_key: int, d_query: int):
        # d_model: dimension of the model (number of features)
        # d_k: dimension of the key (max(dim_model // num_heads, 1))
        # d_q: dimension of the query (max(dim_model // num_heads, 1))

        super().__init__()

        self.query = nn.Linear(d_model, d_query)
        self.key = nn.Linear(d_model, d_key)
        self.value = nn.Linear(d_model, d_key)

    def forward(self, query: tensor, key: tensor, value: tensor, mask: bool = False) -> tensor:

        # Step 1: Apply the linear layers to the query, key and value
        query = self.query(query)
        key = self.key(key)
        value = self.value(value)

        # Step 1.1: Generate the attention mask
        # dimensions of the mask [batch_size, sequence_len, sequence_len]
        if mask:
            mask = generate_attention_mask((query.size(0), query.size(1), key.size(1)))

        # Step 2: Apply scaled dot product attention
        step_2 = scaled_dot_product_attention(query, key, value, mask)

        return step_2
```

Code snippet 3: Attention head

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

    # https://pytorch.org/docs/stable/nn.html#torch.nn.ModuleList
    # https://pytorch.org/docs/stable/generated/torch.cat.html

    def __init__(self,
                 num_heads: int,
                 d_model: int,
                 d_key: int,
                 d_query: int,
                 mask: bool = False):
        # d_model: dimension of the model (number of features)
        # d_k: dimension of the key (max(number of features // num_heads, 1))
        # d_q: dimension of the query (max(number of features // num_heads, 1))

        super().__init__()

        self.mask = mask

        # Step 1: Create the attention heads (so many as num_heads)
        self.attention_heads = nn.ModuleList([AttentionHead(d_model, d_key, d_query) for _ in range(num_heads)])

        # Step 3: Create a linear layer to combine the heads
        self.linear_layer = nn.Linear(d_model, d_model)

    def forward(self, query: tensor, key: tensor, value: tensor, mask: bool = False) -> tensor:
        # Step 1: Apply the attention heads to the query, key and value and concatenate the results
        step_1 = cat([h(query, key, value, self.mask) for h in self.attention_heads], dim=-1)

        # Step 2: Apply the final linear layer to the concatenated results
        step_2 = self.linear_layer(step_1)

        return step_2
```

Code snippet 4: Multi head attention layer

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

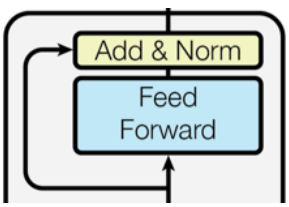



Figure 12: Schematic description of the last FFN in the encoder layer[23]

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

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

```
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
```

Code snippet 5: Last layer of the encoder

In the training process the backpropagation algorithm updates the weights of each linear layer in each encoder layer, starting with the last FFN up to the linear layers that compute the Queries, Keys and Values matrices.

Suppose an architecture with two encoder layers and two multi-head attention layers each, the delta errors and hence the gradients, propagates as follows:

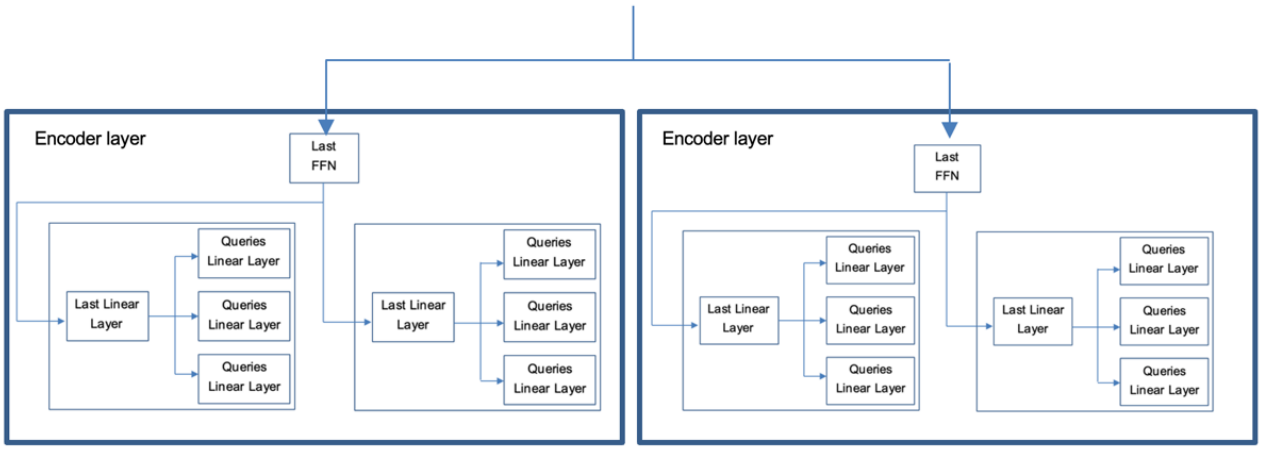



Figure 13: Schematic description of the backpropagation algorithm in the encoder layer

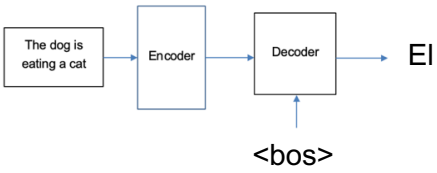
```python
class TransformerEncoderLayer(nn.Module):

    def __init__(self,
                 d_model: int,
                 num_heads: int,
                 units_hidden_layer: int,
                 dropout: float = 0.1,
                 activation: nn.Module = nn.ReLU(),
                 mask: bool = False):
        # d_model: dimension of the model (number of features)
        # num_heads: number of attention heads in the multi-head attention layer
        # units_hidden_layer: number of units in the hidden layer of the feed forward layer
        # dropout: dropout probability

        # d_k: dimension of the key (max(number of features // num_heads, 1))
        # d_q: dimension of the query (max(number of features // num_heads, 1))

        super().__init__()

        dim_q = dim_k = int(div(d_model, num_heads, rounding_mode='trunc'))

        self.attention = Residual(
            MultiHeadAttentionLayer(num_heads=num_heads, d_model=d_model, d_query=dim_q, d_key=dim_k, mask=mask),
            dimension=d_model, dropout=dropout)

        self.feed_forward = Residual(
            feed_forward_layer(d_model=d_model, units_hidden_layer=units_hidden_layer, activation=activation),
            dimension=d_model, dropout=dropout)

    def forward(self, src: tensor) -> tensor:
        # Step 1: Apply the attention layer
        step_1 = self.attention(src, src, src)

        # Step 2: Apply the feed forward layer
        step_2 = self.feed_forward(step_1)

        return step_2
```

Code snippet 6: Transformer encoder layer

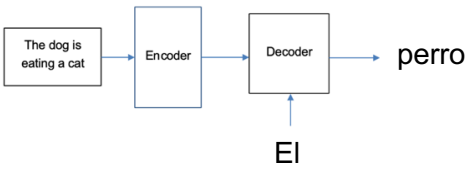
### 2.2.3 Transformer decoder layer

As with the encoder layer, a transformer model can have one or more decoder layers. Its task is to output a d-dimensional vector with the model's result values. 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 "zero" position. In an NLP task, for example, the first target would be an empty token, which means that the sentence begins. Once a one word is computed, that word is used as input to compute the second word. A sentence consisting of both words is used for the third word, and so on. For example, if the task is to translate the sentence "The dog is eating a cat" into Spanish, the process would look like this:

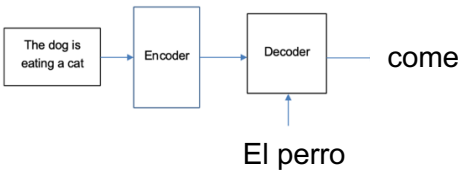
1st Step:



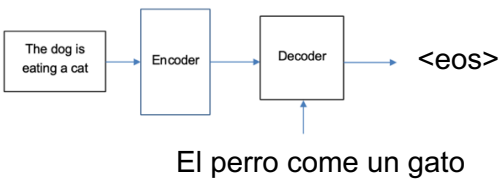

2nd step:




3rd step:




And so on, till the last sequence value.



In detail, the decoder layer works as follow:

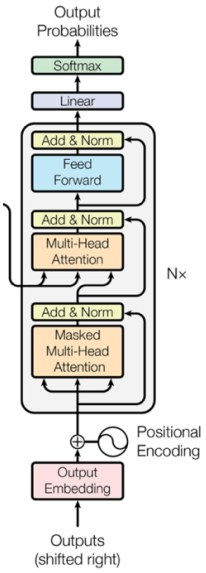



Figure 14: Schematic description of the decoder layer [24]

The target sequence goes through the same self-attention mechanism as in the encoder layer, but here the mask has a different task. The masking task in the decoder is to prevent the sequence from looking 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 those of the self-attention matrix, and its values are -infinity for the values above the diagonal, and 0 for the values below the diagonal and the diagonal itself. This matrix is added to the attention matrix, which leads to the following process:

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

With these dimensions we generate this matrix:

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

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

This matrix is then added to the attention matrix, which is the product of the similarity between the Queries matrix and the Keys matrix. The result is a matrix in which all values above the diagonal are -infinty, and the rest of the values remains invariant.
It could look like this, for example:

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

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 that involves 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 matrix resulting from the decoder self-attention mechanism are the next values of the sequence.
Schematically, it looks like this:

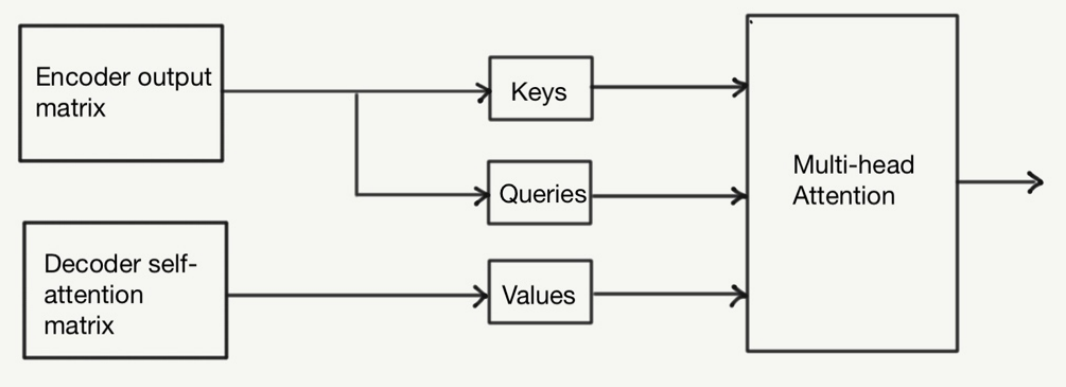



Figure 15: Schematic description of the 2nd Attention layer in the decoder

After this process, the remaining 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. The backpropagation algorithm also propagates in the same way as in the encoder layer.

### 2.2.4 Last steps of the Transformer model

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

## 2.3 Known implementations

Since the advent of “Attention is all you need” in December 2017, many teams have developed their own implementations of the transformer model. The best known and most used are the PyTorch[25] and Tensorflow[26] libraries. Many large models based on the transformer model have been developed.

Some of them are for example:

- BERT[27]: Is a language modelling and next sentence prediction transformer model, developed by Google in 2018. It follows almost exactly the implementation of the original paper. Nowadays, it is one of the most widely used baseline models for NLP experiments.

- GPT-3[28]: The third generation of the GPT series is a pretrained transformer developed for NLP tasks. Has a capacity of 175 billion parameters, and has been trained with more than 500 billion tokens.

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

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

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

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

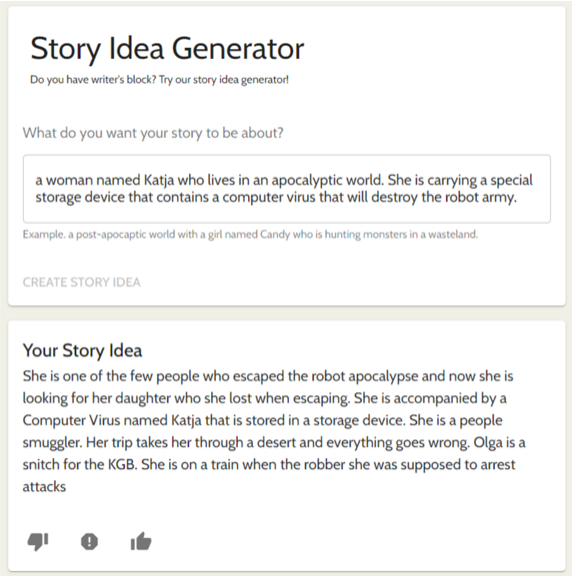



Figure 16: Text generated by GPT-3[29]

- DALL-E[30]: Based on GPT-3, DALL-E is an implementation of a Transformer that interprets natural language inputs and generates corresponding images. It has 12-billion parameters and has been trained on 400 million text-image pairs.


Figure 17: Example of images generated by DALL-E[31]

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

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

[31] 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 dataset consisting of 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 describe the process I followed, starting with the data analysis, and then the actual development and coding of the model. I will show the results obtained and some improvements based on experiments with the first model implementation.

## 3.1 Nature of the data. Data acquisition and preparation.

Any learning process, whether it is a human or a machine process, is always based on data. A child learning to distinguish a dog from a cat must see a real cat or dog (or pictures of them) many times before it is able to do so. The same is true for 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 undertake is to know, analyze and understand the available data.
The data I used in this project are electrical measurements in a factory in Germany. Which factory it is, where it is located and what kind of machines are used there is irrelevant for the purposes of this thesis.

### 3.1.1 Nature of the data and Data acquisition

The dataset consists of 18 different features (19 if we count the timestamp). All of them are electrical measurements collected by IOT sensors and registered in a database.
For this thesis I received the data already registered in a .csv file. I have no further information about the process of obtaining the information, neither how the data adquisition process works, nor how the data is sent and stored in the database.

In this project I use18 different measurements, which are described by 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”[32]. Measured in watts.
  In this case, the measurement reflects the instantaneous 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 in potential energy between two points in an electrical circuit[33].
  In this case, the measurement reflects the difference between one of the three phases and the neutral wire.
  ULL1: Current Voltage between phase 1 and phase 2 wire. In Volts (V)
  ULL2: Current Voltage between phase 1 and phase 3 wire. In Volts (V)
  ULL3: Current Voltage between phase 2 and phase 3 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

[32] https://www.electronics-notes.com/articles/basic_concepts/power/what-is-electrical-power-basics-tutorial.php

[33] 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”[34]. In other words, it is the measurement on how effective the electricity is used in our circuit. Ideally, it 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[35], 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 short-circuit[36].
  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)

The sensors take a measurement every minute and send the data to the database.
The total dataset I received for this project contains 63360 samples, each with 18 measurements and a timestamp. There are some NaN values in the dataset that should be taken care of.

[34] https://www.laurenselectric.com/home/business/understanding-power-factor/

[35] https://www.fluke.com/en/learn/blog/electrical/what-is-frequency

[36] https://www.elandcables.com/the-cable-lab/faqs/faq-what-is-fault-current

### 3.1.2 Data preparation

It is not always possible to use the raw data. The data might be in a format that is not readable, or it might contain some values that do not fit in the set. That could make it difficult to manipulate. Therefore, they usually must be processed in some way.

To edit, analyse and use the data I got in the .csv file, I used the pandas' package[37], which is currently known as the state-of-the-art package for working with data frames and big amounts of data in python[38].

First, I had to parse the entire .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 |

Code snippet 7: Loading dataset into pandas

Is important to check for errors during the data acquisition process. These are displayed in the .csv file as NaN values. These values could be a problem for the learning model; therefore, they should be properly replaced. There are numerous ways to deal with this problem[39], the most common being:

- 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.

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

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

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

My first intuition was to replace the NaN values with the mean of the previous and the next value. But after the first implementation I realised that this solution throws an exception when one (or more) NaN values follow a NaN value. So for this project I decided to replace the NaN values with the previous value. Normally, electrical data should not change much in one minute, therefore, in most situations the wrong value should most likely be similar to the value from a minute ago.

The provided dataset contained 2546 NaN measurements (with a total of 45828 NaN values) out of 63360 total measurements. This represents 4.02% of failed measurements. In my opinion, this is a large number and should be addressed in the future, but it is not part of this thesis, therefore, I will not go into it further.

To replace the NaN values, I used the fillna function[40] form the pandas' package. After this process, I had a clean dataset with 63360 entries of 19 features (one of the features is the measurement timestamp).

```
# Replace all mising values with NaN
dataset = dataset.replace(' ', np.nan)
# Search for all the rows with NaN values
nan_values = dataset[dataset.isna().any(axis=1)]
# Print the shape to know how many are there
print(f'Number of rows with NaN values before cleaning: {nan_values.shape[0]}')

# Fill all NaN values with the previous row value
dataset_clean = dataset.fillna(method='ffill')

# Check that there isn't any NaN values
nan_values = dataset_clean[dataset_clean.isna().any(axis=1)]
# Print the shape to know how many are there
print(f'Number of rows with NaN values after cleaning: {nan_values.shape[0]}')

#Total number of samples
print(f'Total number of samples: {dataset_clean.shape[0]}')
print(f'Number of features: {dataset_clean.shape[1]}')
```

```
Number of rows with NaN values before cleaning: 2546
Number of rows with NaN values after cleaning: 0
Total number of samples: 63360
Number of features: 19
```

Code snippet 8: Filling NaN values

[40] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html

After this process, I had a dataset ready to be analysed.

### 3.1.3 Exploratory Data analysis

Before using the data to train any machine learning model, it is useful to conduct an analysis of the data. Machine learning and data analysis go “hand in hand”. A good data analysis helps us to understand the data we are working with and help us to choose the right model and set the right parameters and hyperparameters[41] for our learning model.

Every minute a measurement is taken and stored; therefore, 1440 samples were taken every day. The entire dataset consists of 63360 samples spread over 44 days (1440 x 44 = 63360). The first measurement was taken on the 1st of June 2020 and the last one on the 14th of July 2020.

#### 3.1.3.1 Scale of the features values

First, I calculate, for every feature, the maximum and minimum value, the median and the mean as well as the standard deviation. In this way, we get comprehensive information about how large is the data range we are dealing with.

```
print_data = False

if print_data is True:
  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('')
```

Code snippet 9: Computing description of data values

[41] https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/

I obtained the following results:

| Feature | min value | max value | mean | median | std. deviation |
|---|---|---|---|---|---|
| PLN1 | 1136.62 | 26071.96 | 7185.27 | 4370.43 | 5619.40 |
| PLN2 | 330.30 | 16792.39 | 4645.44 | 4504.58 | 3948.19 |
| PLN3 | 465.20 | 14512.84 | 4081.81 | 3310.40 | 3423.48 |
| ULL1 | 384.12 | 418.85 | 398.57 | 399.08 | 4.60 |
| ULL2 | 386.56 | 420.71 | 400.63 | 401.10 | 4.38 |
| ULL3 | 384.43 | 418.10 | 398.19 | 398.70 | 4.19 |
| COS_PHI1 | 0.19 | 1.0 | 0.94 | 0..99 | 0.07 |
| COS_PHI2 | 0.38 | 1.0 | 0.91 | 0.99 | 0.12 |
| COS_PHI3 | 0.42 | 1.0 | 0.91 | 0.96 | 0.21 |
| FREQ | 49.86 | 50.12 | 50.0 | 50.0 | 0.02 |
| RC_DC | 1.0 | 11.0 | 4.55 | 5.0 | 0.67 |
| RC_AC | 37.0 | 178.0 | 79.48 | 80.0 | 18.21 |
| RC_50Hz | 3.0 | 14.0 | 8.38 | 8.0 | 1.50 |
| RC_150Hz | 18.0 | 47.0 | 32.32 | 32.0 | 6.71 |
| RC_<100Hz | 16.0 | 45.0 | 28.91 | 28.0 | 6.20 |
| RC_100Hz-1kHz | 36.0 | 166.0 | 74.09 | 75.0 | 17.14 |
| RC_1kHz | 26.0 | 172.0 | 68.77 | 68.0 | 17.71 |
| RC_>10kHz | 0.0 | 11.0 | 3.77 | 3.0 | 1.51 |

Figure18: Table with description of the data

These results show that there is a big difference in the scale of the different features. Some features, like PLN1 are in the scale of tens of thousands (between 1000 and 26000), while others, like COS_PHI, are between 0 and 1. This could be a problem for training the model and would require some kind of normalisation and standardisation[42].

Of all the different normalisation procedures, I have chosen to do a mean standard deviation normalisation.
The mean value of this feature value is subtracted from the original value and the result is divided by the standard deviation of this feature.

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

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

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
```

Code snippet 10: Applying data normalization

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

Code snippet 11: Data sample before normalization

Same 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 |

Code snippet 12: Data sample after normalization

### 3.1.3.2 Distribution of the data

Then I computed some plots to visualize the nature of the data and its distribution. I calculated two types of charts for each feature. A line plot with the data of a whole week, and a boxplot for each week.
The first plot is the feature value in a time axis, the second is a boxplot showing the distribution of the data of each week:
(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):

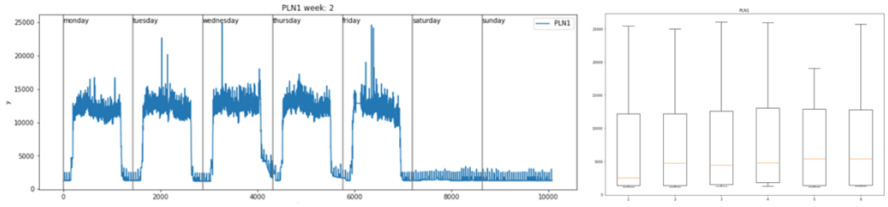



Figure 19: Plots with distribution of PLN feature

- ULL1, ULL2, ULL3 (Same as above, I am only showing ULL1):

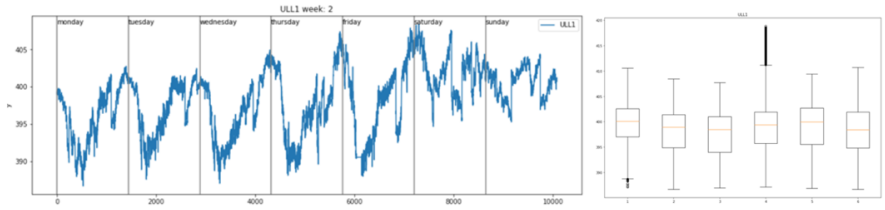



Figure 20: Plots with distribution of ULL feature

- COS_PHI1, COS_PHI2, COS_PHI3 (Same as above, only showing COS_PHI2):

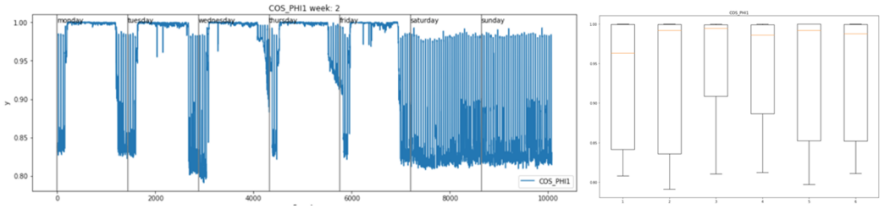



Figure 21: Plots with distribution of COS_PHI feature

- FREQ:

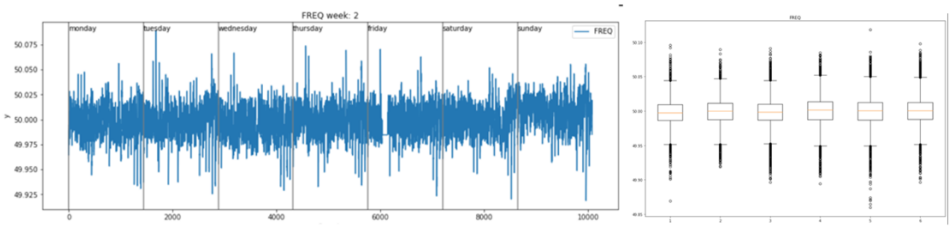



Figure 22: Plots with distribution of COS_PHI feature

- RC_DC:

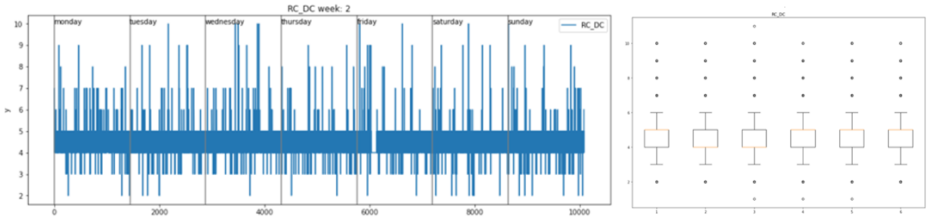



Figure 23: Plots with distribution of COS_PHI feature

- RC_AC:

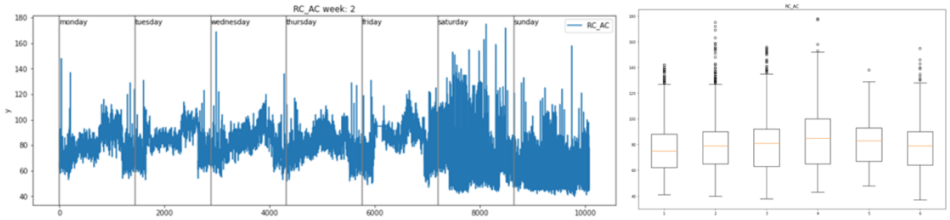



Figure 24: Plots with distribution of COS_PHI feature

The rest of features (RC RC_50Hz, RC_150Hz, RC_<100Hz, RC_100Hz – 1KHz, RC_>100KHz) are very similar to RC_AC in its distribution. Detailed values and plots of these features can be found in the Git repository[43] accompanying this Thesis.

### 3.1.3.3 Correlation between features

Another important analysis is to see if there are correlations between the different features.

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

In this case, I was not dealing with ranked features. Therefore, I used the Pearson's correlation coefficient[44].

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$$

- Cov(X,Y): Covariance between feature X and feature Y
- $\sigma_X$: Variance of the feature X
- $\sigma_Y$: Variance of the feature Y

```
correlations = []
matrix = []

for i in dataset_norm.columns[1:]:
  feature = []
  for j in dataset_norm.columns[1:]:
    print(f'Correlation between {i} and {j}')
    correlation = stats.pearsonr(dataset_norm[i], dataset_norm[j])[0]
    if i != j:
      correlations.append(abs(correlation))
      feature.append(abs(correlation))
      print(correlation)
  print(f'Mean of {i} correlations: {np.mean(feature)}')
  print('')
  matrix.append(feature)

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

Code Snippet 13: Computing the correlation between features

After calculating the correlations, I could see that there is generally a large correlation between the features. The mean value of all correlations (17 x 17, 289 different correlations) is 0.457. In order to look at the features correlation individually, I plot the results in a heat map:

[44] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

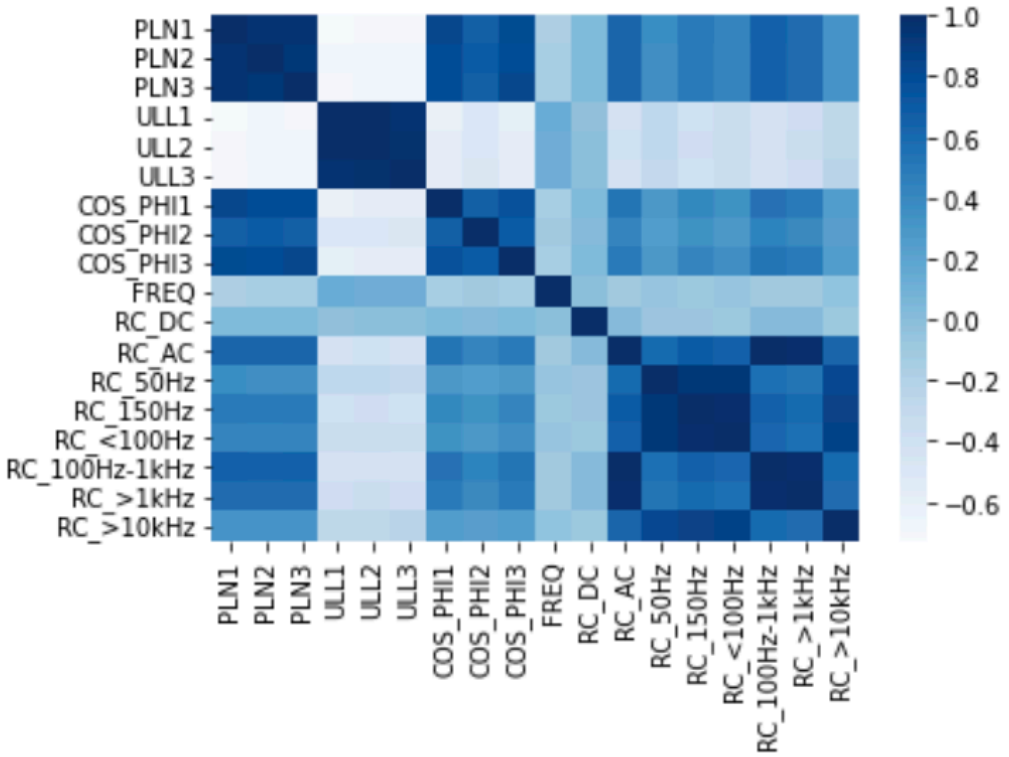



Figure 25: Feature correlations heat map

Plotting the results of the correlations between the features individually, I obtained these plots:

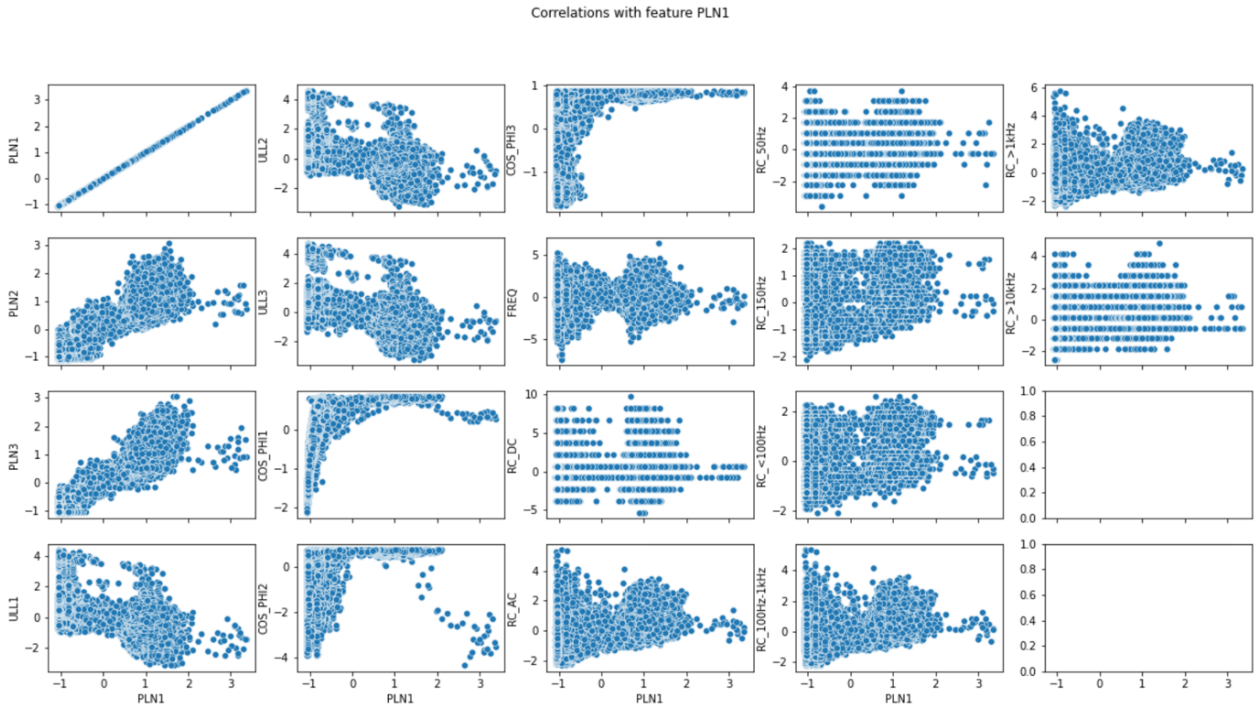



Figure 26: Correlations plot PLN feature

(The correlations of PL2 and PLN3 are similar to PLN1, therefore I am not going to show it hier. If the reader is interested in the plots of those features, they could be found in the project's github repository[45])

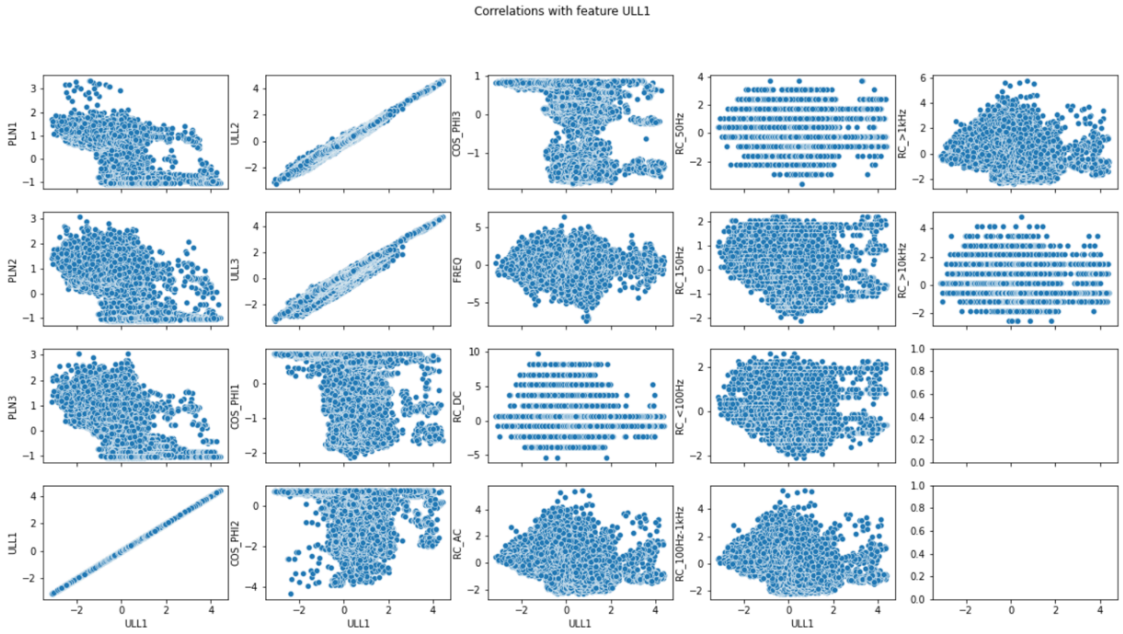



Figure 27: Correlations plot ULL feature

(The correlations of ULL2 and ULL3 are similar to ULL1, therefore I am not going to show it here. If the reader is interested in the plots of those features, they could be found in the project's github repository[62])

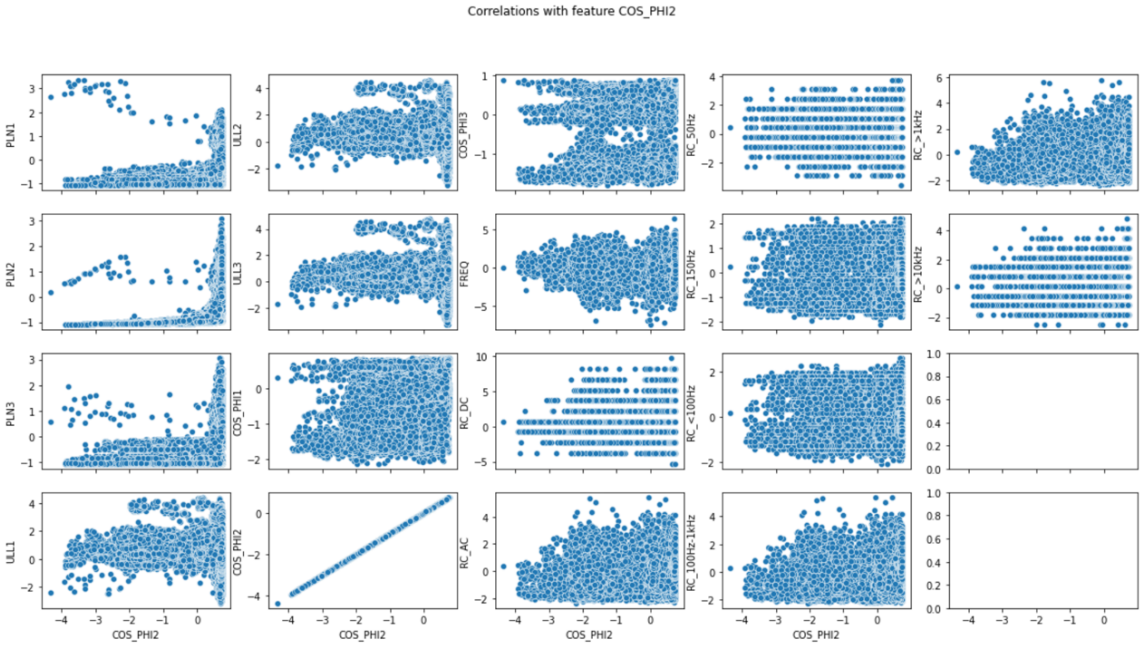



Figure 28: Correlations plot COS_PHI feature

[45] https://github.com/SergioTallo/Bsc_Thesis

(The correlations of COS_PHI1 and COS_PHI3 are similar to COS_PHI2, therefore I am not going to show it here. If the reader is interested in the plots of those features, they could be found in the project's github repository[46])

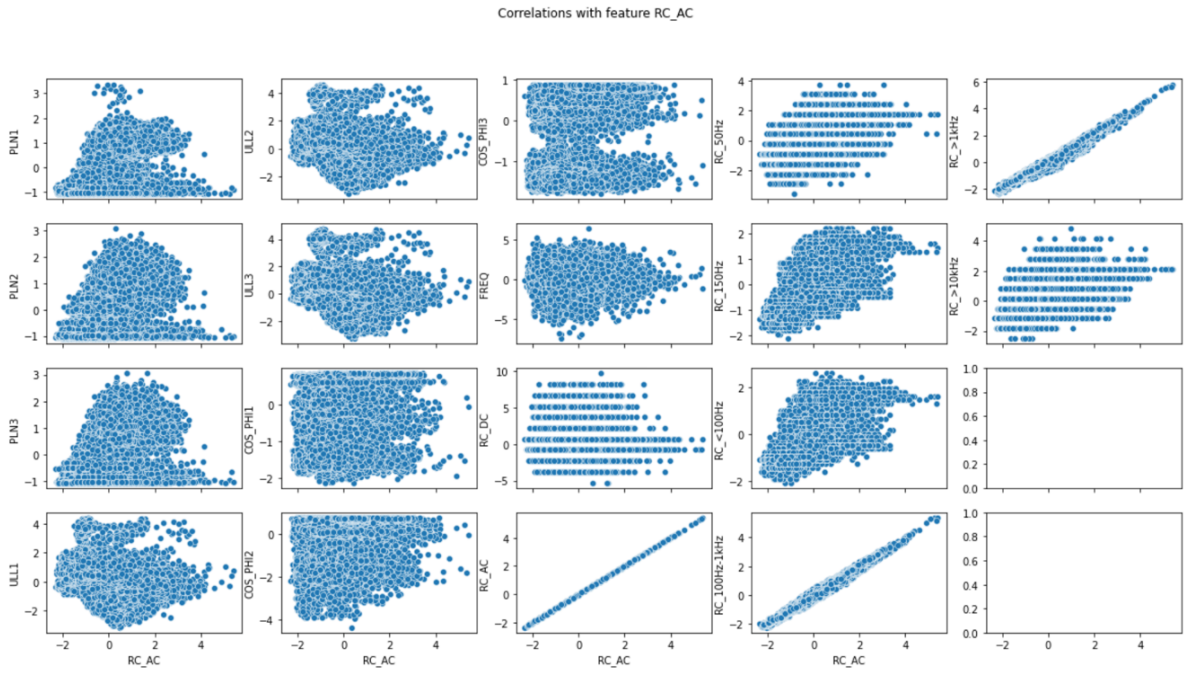



Figure 29: Correlations plot COS_PHI feature

(The correlations of RC_50Hz, RC_150Hz, RC_<100Hz, RC_100Hz-1kHz and RC_>10kHz: are similar to RC_AC, therefore I am not going to show it here. If the reader is interested in the plots of those features, they could be found in the project's github repository[64])

(The features FREQ and RC_DC are strongly uncorrelated to the rest of the features, therefore I considered not interesting to show the correlation plots of both these features. If the reader is interested in the plots of those features, they could be found in the project's github repository[64])

In general, we can see that all the features have a very low correlation with FREQ and RC_DC. The rest of the features can be grouped in PLN, ULL, COS_PHI and two groups for RC. This observation is in line with the logical expectations as these feature groups correspond to similar physical observations.

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

[46] https://github.com/SergioTallo/Bsc_Thesis

```python
# 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} principal component: {eigen/np.sum(eigenvalues)} (accumulated {round(acc, 4)})')
```

```
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: 0.0002182637426103455 (accumulated 0.9994)
Explained_variance 17 principal component: 0.00023351820917083064 (accumulated 0.9996)
Explained_variance 18 principal component: 0.0004035851322244031 (accumulated 1.0)
```

Code Snippet 14: Computing covariance, eigenvalues and explained variance

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

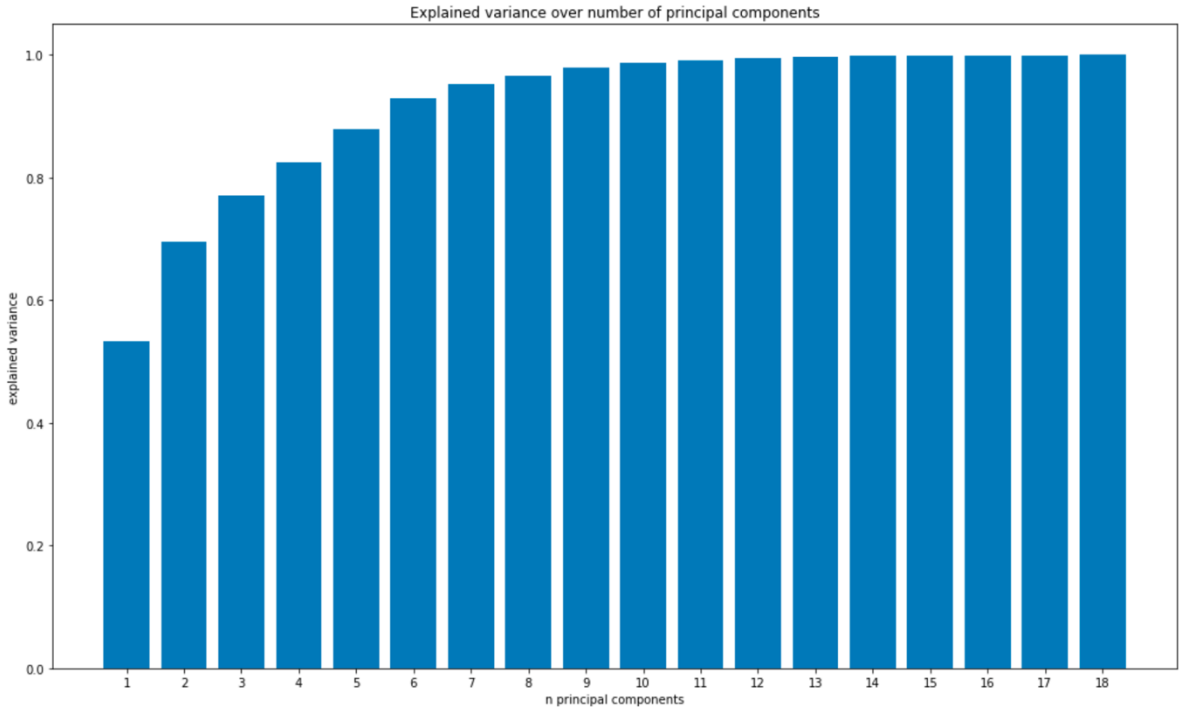



Figure 30: Plot showing the explained variance over the number of principal components

### 3.1.3.4 Final conclusions of the data analysis

After performing the exploration data analysis, I was able to draw two conclusions:
First, I observed two very different time frames:
The first took place on weekdays, between 4:30 and 19:30.

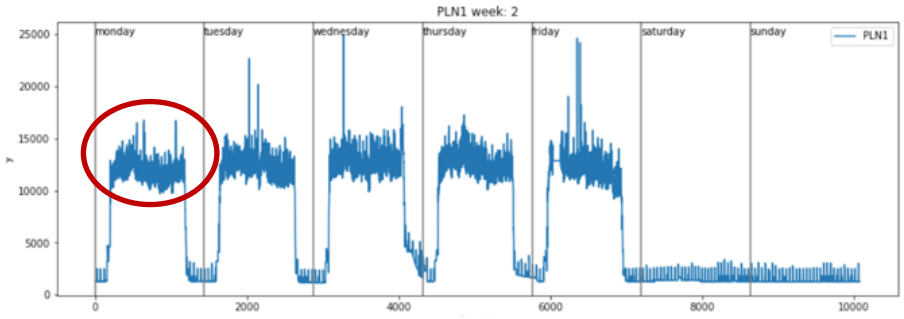



Figure 31: Data distribution weekdays

The second occurred throughout the day on weekends and on weekdays between 19:30 and 4:30.

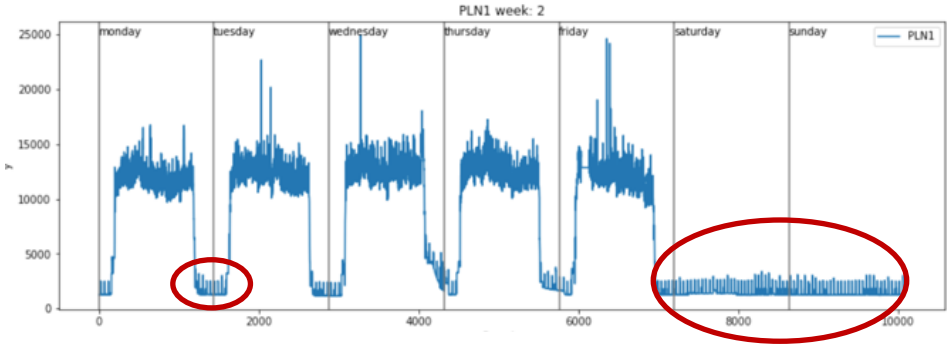



Figure 32: Data distribution weekend

My first intuition was to think that the factory doesn't have any activity during the second time frame. If this were the case, I could simply eliminate these samples, as the focus should be on working hours and data outside this period could only introduce noise into the dataset. But, after checking with my thesis supervisor, I found that the factory is not closed, but has a completely different activity. Therefore, I kept using those values as part of the data frame. In addition, further experiments in the early stages of this project showed the model training results were no better when using the whole dataset than when using only one of these time frames. Therefore, this initial intuition was discarded for the two reasons mentioned above.

The second 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 properly. It also shows that any dimensionality reduction technique (e.g., PCA or t-SNE) could be useful, even some features could be redundant[47]. Nevertheless, we must keep in mind that any dimensionality reduction leads to a loss of information.

In the case of our dataset, I do not believe that any kind of dimensionality reduction or feature selection is needed for the initial stages training. There are only 18 features, a number that is easily handled by any computer nowadays, so we can train the model with all features without losing any information. However, to improve the results of the training or to gain new insights, it might be useful in the future to reconsider the idea that most of the variance lies within the first 6 principal components. Specially to avoid overfitting if the model show signs of it.

## 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 predict the next element of a sequence of 30 measurements, having 30 consecutive measurements and a new element as input.

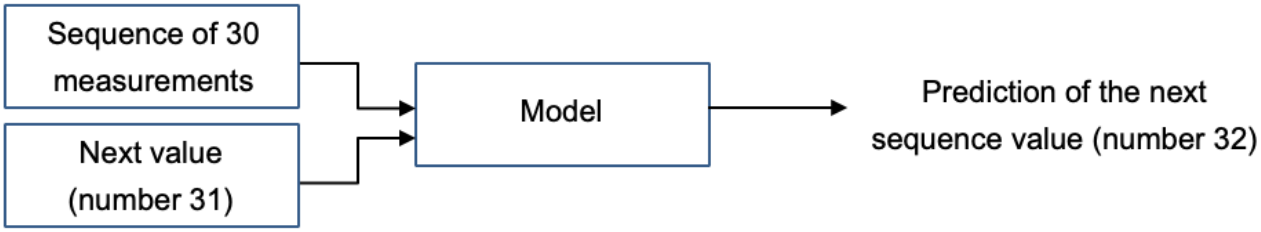



Figure 33: Schematic description of the prediction process

The model is a standard implementation of an encoder-decoder transformer architecture using the PyTorch library[48] [49] [50] [51], with a self-implementation of the mask and the positional encoding.

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

[48] https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoder.html

[49] https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoderLayer.html

[50] https://pytorch.org/docs/stable/generated/torch.nn.TransformerDecoder.html

[51] https://pytorch.org/docs/stable/generated/torch.nn.TransformerDecoderLayer.html

```python
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(self, 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)))

        position_encoding = position_encoding.to(device)

        return position_encoding
```

```python
    def forward (self, enc_input, dec_input):

        memory_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=None, memory_mask=None)

        return output
```

Code Snippet 15: Implementation of transformer model

### 3.2.1 Training and testing

The entire training process was implemented in a Jupyter Notebook using a pro subscription to Google Colab[52] with a Tesla P100-PCIE-16GB as GPU.
The model does not accept the raw dataset in pandas' format. Before the model can be fed with the data, the data must therefore to be prepared.

#### 3.2.1.1 Preparing the data for training

I used Pytorch data loaders[53] to feed the models.
First, I had to create tuples of inputs (encoder input and decoder input) and their corresponding target. I chose a sequence length of 30. Therefore, I chose sequences of 30

[52] https://colab.research.google.com

[53] https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

consecutive samples (an explanation of this choice will be shown in the chapter 3.2.2) as the encoder input, the next sample in the sequence (sample number 31) as the decoder input, and the next sample (sample number 32) as the target.

For example, if we have the 30 consecutive samples starting with timestamp 2020-06-01 00:00:00 and ending with timestamp 2020-06-01 00:00:29 as encoder input, 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 the 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 to achieve 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 the decoder input and the following one as the target. And start the process again, but with 1 as the starting point.
Following this strategy, we obtain 63330 pairs of sequences with a length of 30 samples.

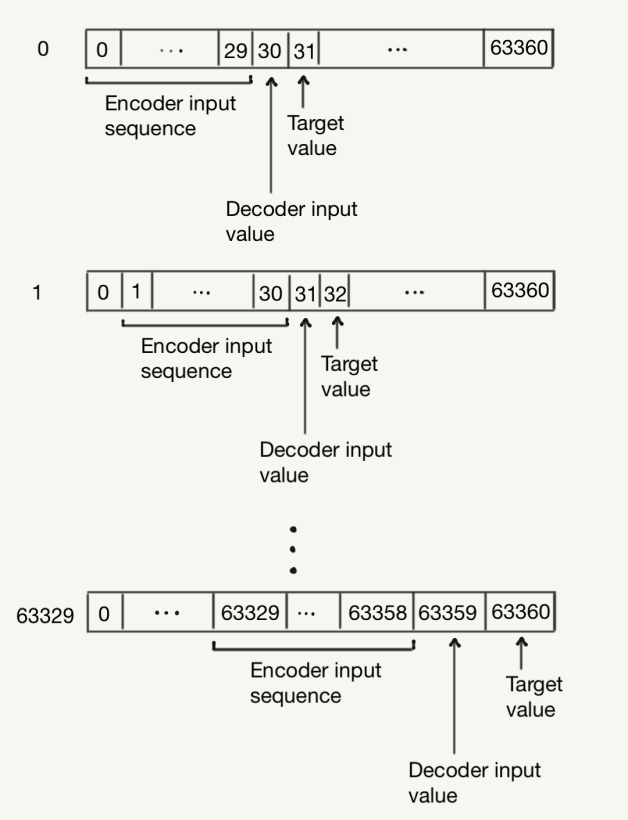



Figure 34: Schematic description of the algorithm to make training pairs

```python
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) - 31)):
    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)
```

Code Snippet 16: Creating data pairs input/target

The next step in preparing the data for training is to split it into 2 datasets. One to train the model and one to test the training results. For this purpose, I used the train_test_split library from sklearn[54]. With a test_size of 0.1 and shuffle=true to achieve a random split.

```python
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)
```

Code Snippet 17: Splitting into train/test set

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

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

The last step is to create the data loaders. I created two data loaders with mini-batches of 18 sequences. One for training purposes and one for testing the data. Inn both, the data was shuffled. The result is 3563 mini-batches of 16 sequences for the training dataset and 396 batches of 16 sequences for the testing dataset.

```
# 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=16,
    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=16,
    shuffle=True
)

return loader_train, loader_test
```

Code Snippet 18: Creating data loaders with mini batches of data

I chose the size of 16 for the mini-batches following the advice of Yann LeCun and Nitish Shirish Keskar et al. Large batch sizes tend to produce “sharper” local minima, which is a problem because it leads to overfitting[55]. In the words of Yann LeCun (Turing Award in 2018) “Friends don’t let friends use minibatches larger than 32”[56]. Larger batch sizes are better in terms of training time, but worse in terms of error rate. This is a trade-off that should be adjusted differently for each project. Also, the batch sizes should be powers of 2, as powers of 2 make more efficient use of resources on CPUs and GPUs[57].

At this point, I had prepared the data for the model and was ready to start the training.

[55] Keskar, N.S., Mudigere, D., Nocedal, J., Smelyanskiy, M. and Tang, P.T.P., 2016. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*.

[56] https://twitter.com/ylecun/status/989610208497360896?lang=en

[57] Aggarwal, C.C., 2020. Linear Algebra and Optimization for Machine Learning: A Textbook. Springer International Publishing. Page 165.

### 3.2.1.2 Baseline model

The first question that arises when training any model is to answer 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”[58]. The results of our final model should at least be better than the result of our baseline model. This would be a good starting point for further development of our model.
There are several strategies to build a good baseline model. In regression tasks (like the task in this thesis), the most common are:

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

In the first baseline model I used, I used predicting the input as the 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')
```

Code Snippet 19: Implementing first base line model

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

I used the input as the output and compared it to the ground truth and used the mean square error[59] (MSE) as a loss function,

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

I got the following results:

- The mean value of the loss over the entire training dataset is: 0.472
- The mean value of the loss over the entire test dataset is: 0.483

I have also created a second baseline model that uses a normal FFN to train data sequences.
The idea is to train a FFN that outputs a single sample and compares it with the next sample in the dataset. I am aware that I am not looking at the samples as a series, just as individual samples, but I am just using this approach to create a better baseline model.
The model I used was a simple FFN in PyTorch[60]. With 4 hidden layers, an input of 18 dimensions and an output of 18 dimensions, with ReLU[61] as activation function.

```
ANN_relu(
  (linear1): Linear(in_features=18, out_features=180, bias=True)
  (linear2): Linear(in_features=180, out_features=640, bias=True)
  (linear3): Linear(in_features=640, out_features=180, bias=True)
  (linear4): Linear(in_features=180, out_features=18, bias=True)
  (relu): ReLU()
  (dropout): Dropout(p=0.2, inplace=False)
)
```

Code Snippet 20: Description of FFN for baseline model

Trained over 200 epochs using MSE[62] as loss function and Stochastic Gradient Descend with a learning rate of 0.01 and a momentum of 0.9 as the optimisation[63] method.

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

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

[61] https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

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

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

After splitting the data into training and test samples at a ratio of 0.1 and after the training, I obtained the following results:

| Algorithm parameters | MSE Training set | MSE Test set | Training Time (200 Epochs) | Training time per Epoch |
|---|---|---|---|---|
| Baseline model, FFN, 4 hidden layers 200 epochs | 0.272 | 0.288 | 22 minutes | 4 seconds |

Figure 35: FFN Baseline model training results

I created the following graph for comparison with the first base model:

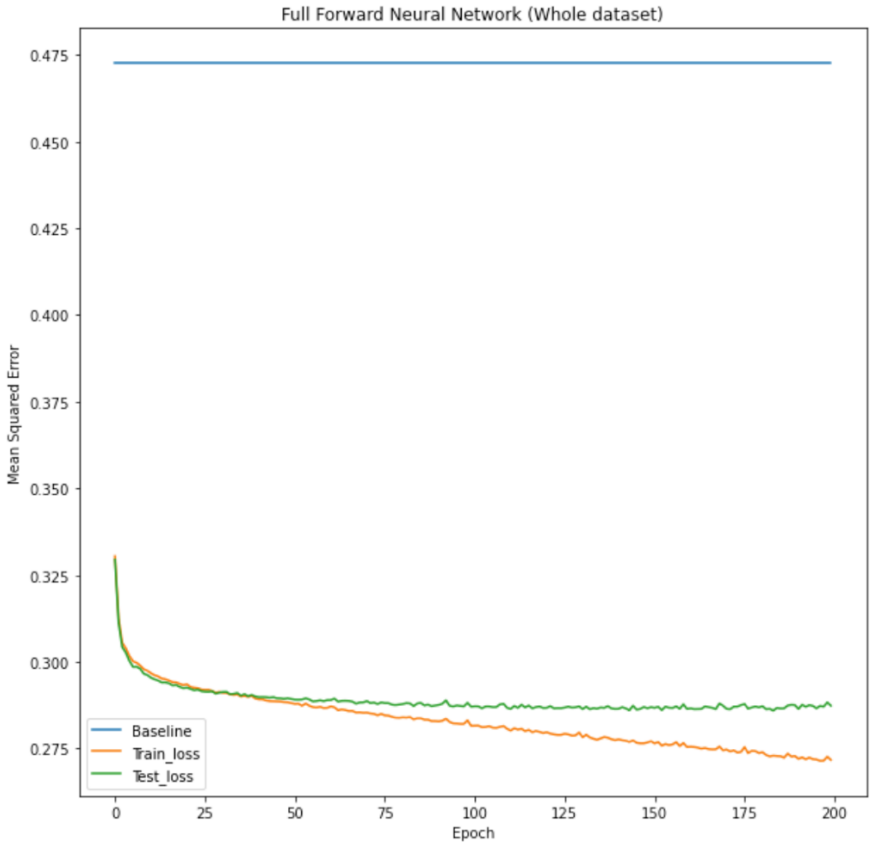



Figure 36: Plot results training FFN for baseline model

This is indeed an improvement over the first baseline model, although it starts to show signs of overfitting around epoch 50. The model shows the best performance in the test set at epoch 185 with a mean loss of 0.285.

These results are better than those of the first baseline model, and our final model should improve them as well. We have to keep in mind that this baseline model is not based on a

well-trained, researched FFN. I am sure that with more time and effort I could improve these baseline results.

### 3.2.1.3 First training of the transformer model

Schematically, one training iteration looks like this:

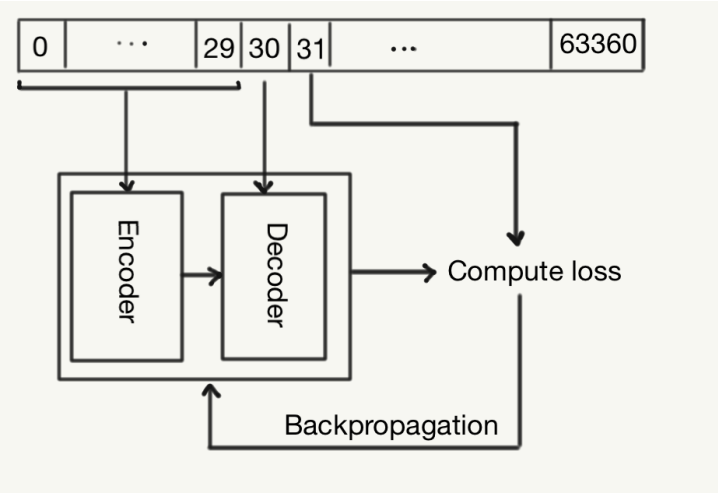



Figure 37: Schematic description of the training process

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

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

```
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=True)
```

Code Snippet 21: Transformer model definition

As optimiser, I chose the PyTorch implementation of Stochastic Gradient Descend[64] with a learning rate of 0.01 and as loss function criterion the PyTorch implementation of Mean Square Error[65].
I trained the model over the previously calculated data loaders and over 200 epochs. I calculate the mean loss over the training dataset and over the test dataset. Both values were saved to track the performance of the training in each epoch. As a reminder, both the training and test datasets were calculated in the previous sub-chapter (3.2.1.1.).
A training epoch works as follow:

- Iterate over the entire train loader (each batch in the loader is a tuple containing the encoder input sequence, the decoder input element, and the ground truth element corresponding to that input, named as target)
- Perform the forward-pass over a batch of data.
- Calculate the MSE with the output of the previous step and the target.
- Perform backpropagation and gradient descend (the gradients are normalised[66])

```
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()
```

Code Snippet 22: Implementation of the training process

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

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

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

After training, I calculated the mean error over the entire training dataset and over the entire test dataset, ran the forward pass, compared it to the target values, and stored the results.

```
  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)

  print('\n')

return model, epoch_loss_train, epoch_loss_test
```

Code Snippet 23: Implementation of the test process

And then just repeat the process the desired number of epochs until the achieved test loss provides a satisfactory result. In this use case, I trained the model over 200 epochs. In doing so, I obtained the following results:

| Algorithm parameters | MSE Training set | MSE Test set | Training Time (200 Epochs) | Training time per Epoch |
|---|---|---|---|---|
| 6 Encoder Layers, 1 Decoder Layer, 6 Heads. SGD. 200 Epochs | 0.242 | 0.258 | 6 hours 30 min. | 83 seconds |

Figure 38: Transformer Vanilla model Training results

For comparison with the FFN baseline model, I created the following graph:

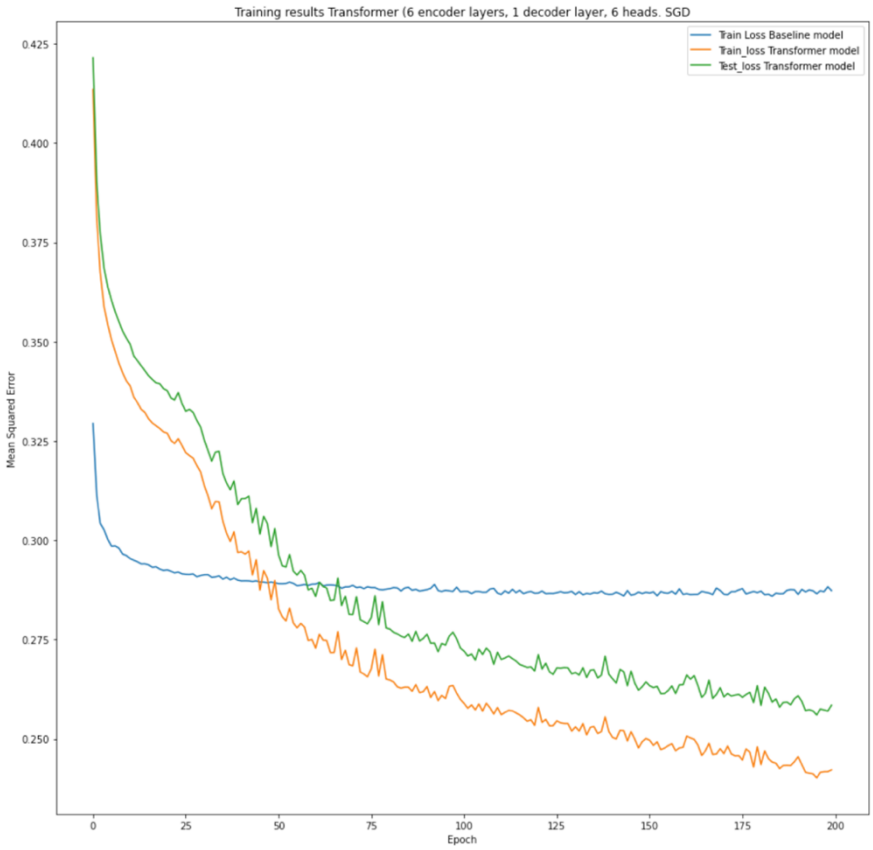



Figure 39: Plot training results Transformer Vanilla model

The results show an improvement compared to both baseline models. This proves that using a transformer model is a valid proof of concept for this task.

### 3.2.2 Improving the model

The first improvement I tried was to change the optimisation algorithm. Instead of an SGD I used the PyTorch implementation of ADAM[67] [68] with a learning rate of 0.001.
Keeping the other hyperparameters and training over the same training and test dataset and over 200 epochs, I obtained the following results and then calculated a graph comparing them to the results of the first training implementation:

| Algorithm parameters | MSE Training set | MSE Test set | Training Time (200 Epochs) | Training time per Epoch |
|---|---|---|---|---|
| 6 Encoder Layers, 1 Decoder Layer, 6 Heads. ADAM. 200 Epochs | 0.304 | 0.315 | 7 hours 30 min. | 100 seconds |

Figure 40: Transformer (ADAM optimizer) Training results

For comparison with the results of the "vanilla" model, I created the following graph:

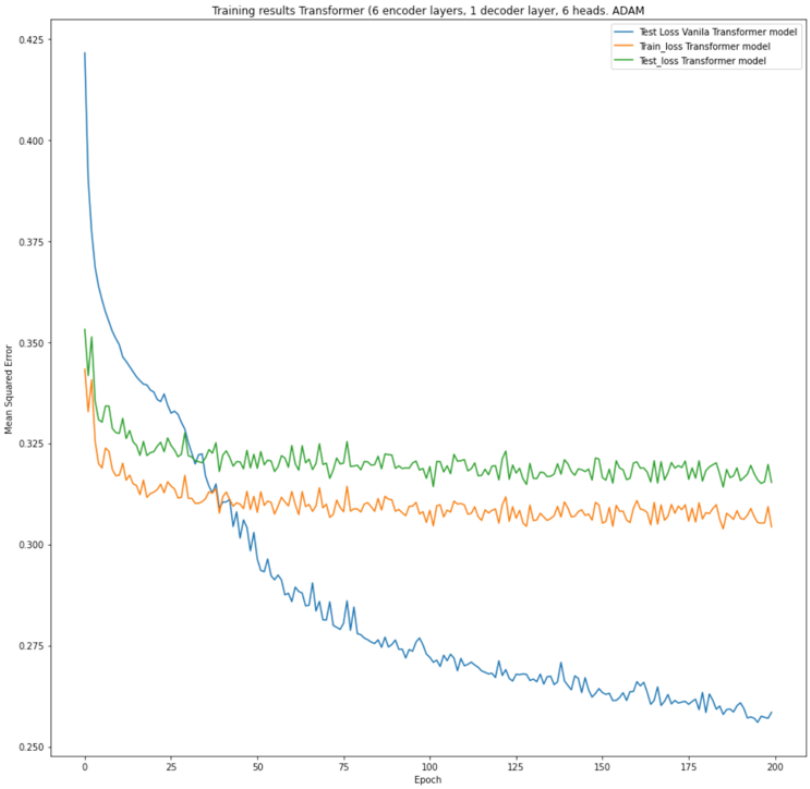



Figure 41: Plot training results Transformer ADAM optimizer

[67] https://pytorch.org/docs/stable/generated/torch.optim.Adam.html
[68] https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

The results of this training showed no improvement over using SGD. Very unstable error rates in both the training and test sets. In this case, is not recommended to use Adam as an optimisation algorithm.

The second improvement I tried was to return to the SGD optimisation algorithm but with the addition a momentum. I repeated the training with the same data, the same hyperparameters and with the PyTorch implementation of SGD, this time with a learning rate of 0.01 and a momentum of 0.9. I get the following results:

| Algorithm parameters | MSE Training set | MSE Test set | Training Time (200 Epochs) | Training time per Epoch |
|---|---|---|---|---|
| 6 Encoder Layers, 1 Decoder Layer, 6 Heads. SGD Mom. 200 Epochs | 0.192 | 0.221 | 8 hours 15 min. | 2 minutes |

Figure 42: Transformer SGD with momentum optimizer training results

For comparison with the results of the "vanilla" model, I created the following graph:

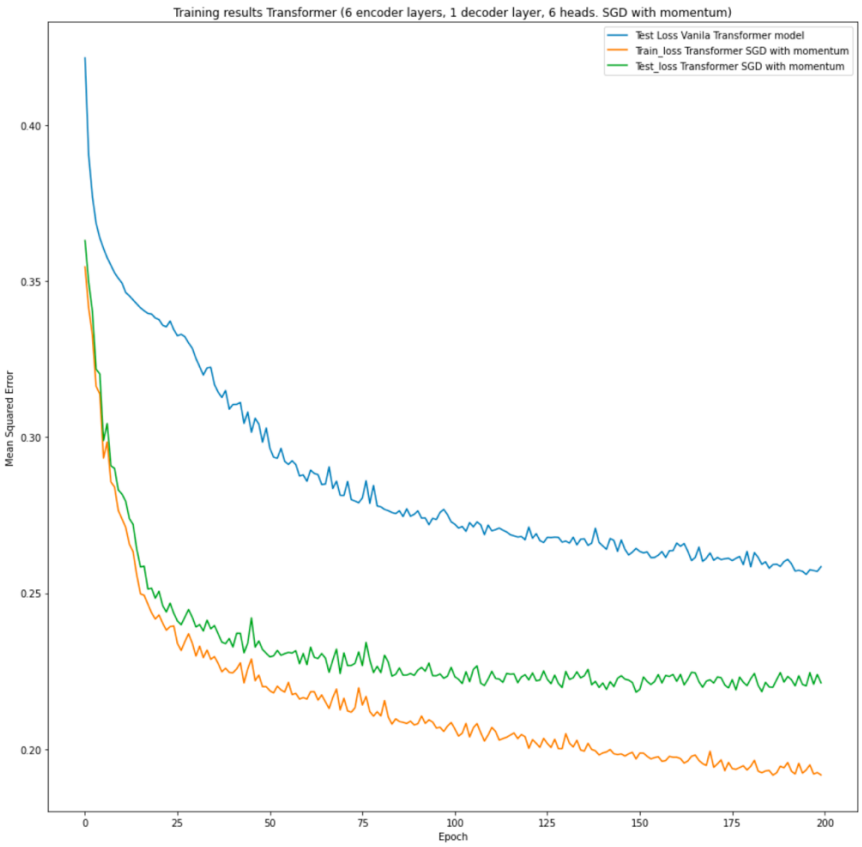



Figure 43: Plot training results Transformer SGD with momentum optimizer

As expected, the results show an improvement when compared to the results of SGD without momentum. It is clearly more stable than ADAM and it shows no signs of overfitting in the first 200 epochs. These results can be improved training the model through more epochs. The training time are worse than when using SGD without momentum, but within a reasonably scale. These results show that, using SGD with momentum is a better choice for the learning process than ADAM or SGD without momentum.

Other possible improvements could be achieved 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 create a small version of the model to see how the results change with the smallest possible model. I implement a model with only 1 encoder layer, 1 decoder layer, and 1 multi-head attention layer. I trained for 200 epochs using SGD with momentum as optimiser and got the following results:

| Algorithm parameters | MSE Training set | MSE Test set | Training Time (200 Epochs) | Training time per Epoch |
|---|---|---|---|---|
| 1 Encoder Layers, 1 Decoder Layer, 1 Heads. SGD Mom. 200 Epochs | 0.215 | 0.232 | 3 hours 15 min. | 38 seconds |

Figure 44: Smaller Transformer training results

For comparison with the results of the "vanilla" model, I created the following graph:

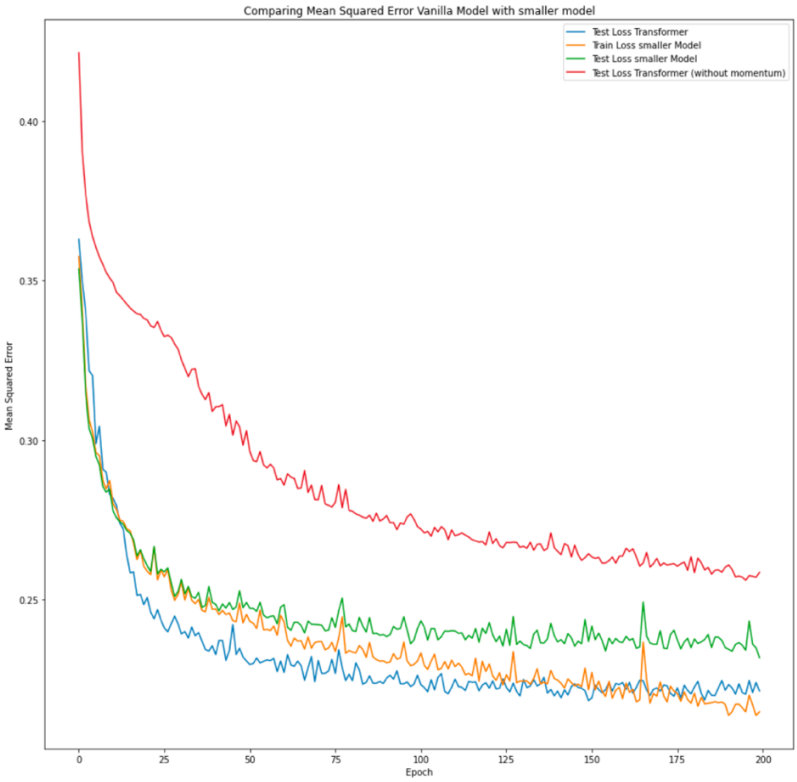



Figure 45: Plot training results smaller Transformer

As expected, the error rate after 200 epochs was higher than with a larger model but using momentum in the optimiser gave better results than a larger model without using momentum. This proves that using SGD with momentum is a better choice than the other optimisers tested. As can be seen from the results, a smaller model mean better time performance. Further research could show what is the better choice in the trade-off between time performance and error results.

As final research model, I implemented a larger model to see if there was a significant improvement in the error results and to compare it to the smaller ones, both in terms of time and error performance. I implemented a model with 10 encoder layers, 5 decoder layers, and 9 multi-head attention layers. I trained it during 200 epochs using SGD with momentum as optimiser and got the following results:

| Algorithm parameters | MSE Training set | MSE Test set | Training Time (200 Epochs) | Training time per Epoch |
|---|---|---|---|---|
| 10 Encoder Layers, 5 Decoder Layer, 9 Heads. SGD Mom. 200 Epochs | 0.193 | 0.218 | 14 hours | 3 minutes |

Figure 46: Bigger Transformer SGD training results

And comparing them with the results of the model using SGD with momentum, I computed the following plot:

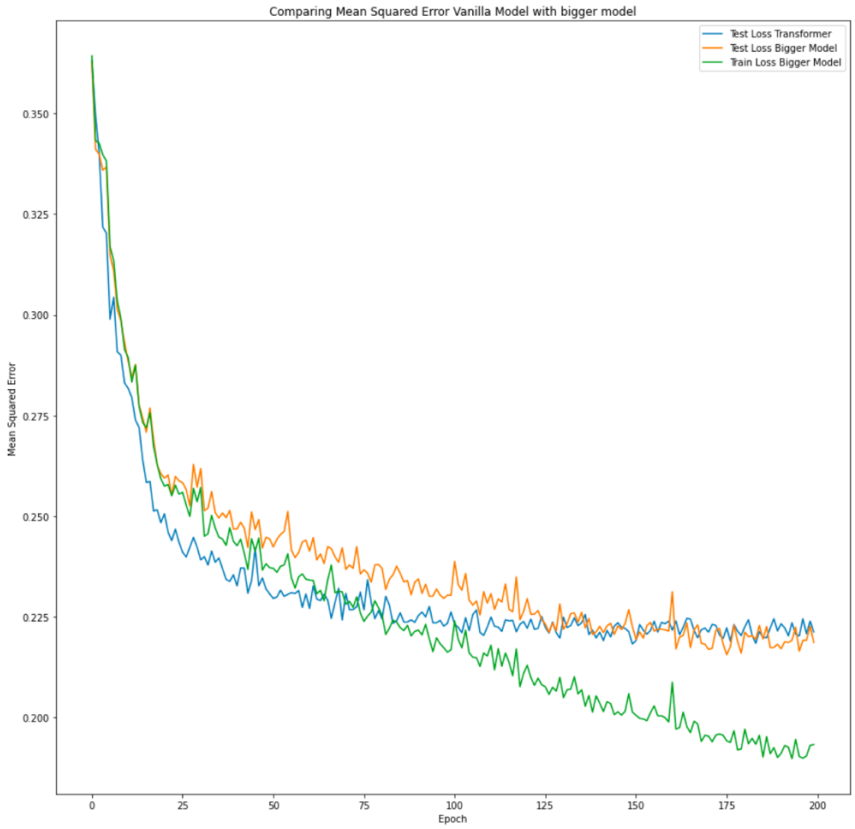



Figure 47: Plot training results bigger Transformer

The results are better than the smaller model results. This prove that more research in changing the hyperparameters could bring better results and that a model with more trainable parameters in many cases improve the error rate. Though, in this case, the improvement is not very significant, the loss in the test set shows in the plot that more epochs could improve the MSE in the test dataset. 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.

For comparison with the results of all other models, I created the following graph:

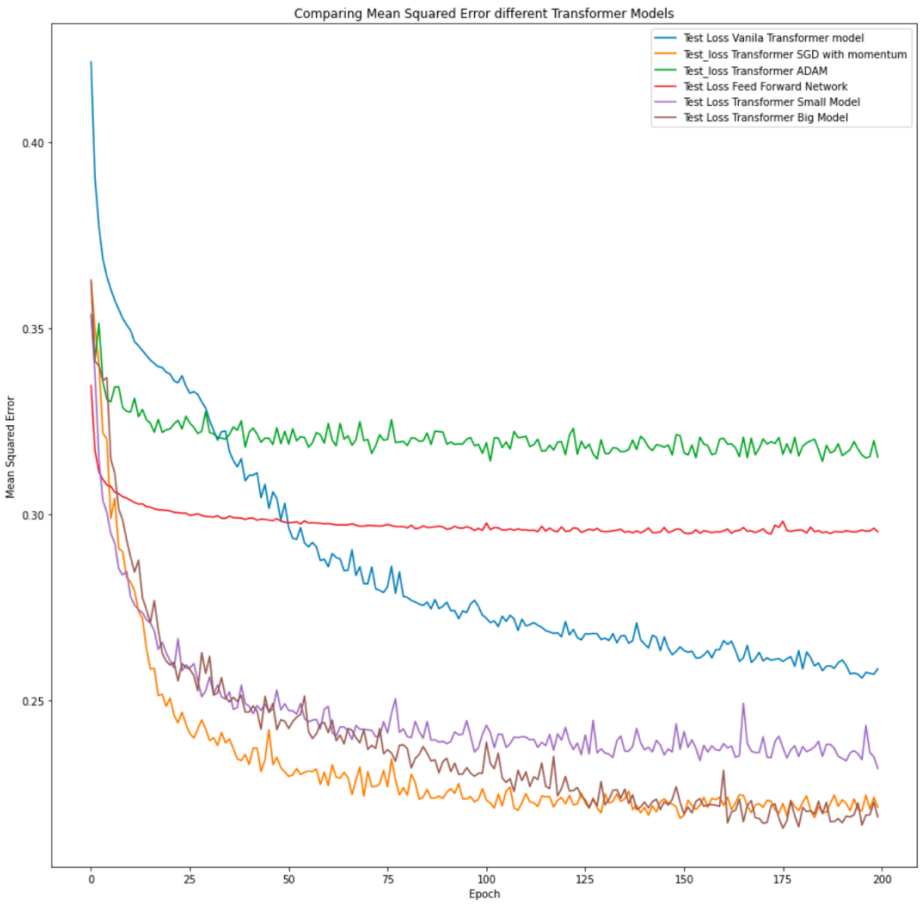



Figure 48: Plot training results comparing different transformer models

I compared the results of the four main models with the number of their parameters and created the following graph:

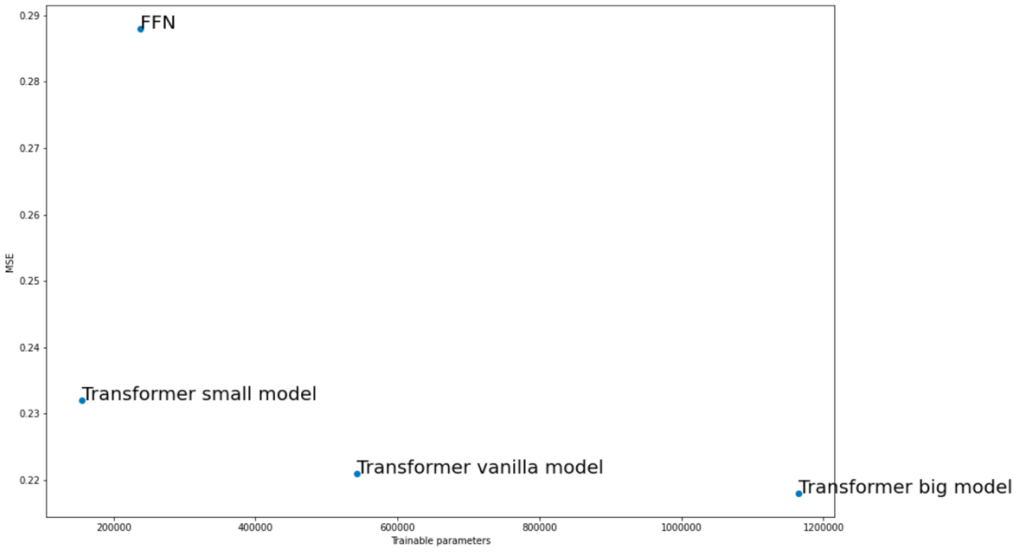



Figure 49: Plot comparing MSE and the number of training parameters

This comparison shows that the transformer model is more efficient than a FFN in terms of error rate per trainable parameter. Between different transformers size, a bigger model, although the error rate is lower, is less efficient.

I also did some experiments where I changed the length of the sequence. I tried with sequence length of 120, 60, 15 and 2. I got the following results:

| Algorithm parameters | MSE Training set | MSE Test set | Training Time (200 Epochs) | Training time per Epoch |
|---|---|---|---|---|
| Sequence Length 120 | | | | |
| 6 Encoder Layers, 1 Decoder Layer, 6 Heads. SGD Mom. 200 Epochs | 0.193 | 0.221 | 13 hours | 150 seconds |
| Sequence Length 60 | | | | |
| 6 Encoder Layers, 1 Decoder Layer, 6 Heads. SGD Mom. 200 Epochs | 0.196 | 0.228 | 8.5 hours | 130 seconds |
| Sequence Length 30 | | | | |
| 6 Encoder Layers, 1 Decoder Layer, 6 Heads. SGD Mom. 200 Epochs | 0.192 | 0.221 | 8 hours | 120 seconds |
| Sequence Length 15 | | | | |
| 6 Encoder Layers, 1 Decoder Layer, 6 Heads. SGD Mom. 200 Epochs | 0.194 | 0.221 | 5 hours | 68 seconds |
| Sequence Length 2 | | | | |
| 6 Encoder Layers, 1 Decoder Layer, 6 Heads. SGD Mom. 200 Epochs | 0.252 | 0.272 | 4.5 hours | 64 seconds |

Figure 50: Training Results using different sequence length

I compared them and created the following graph:

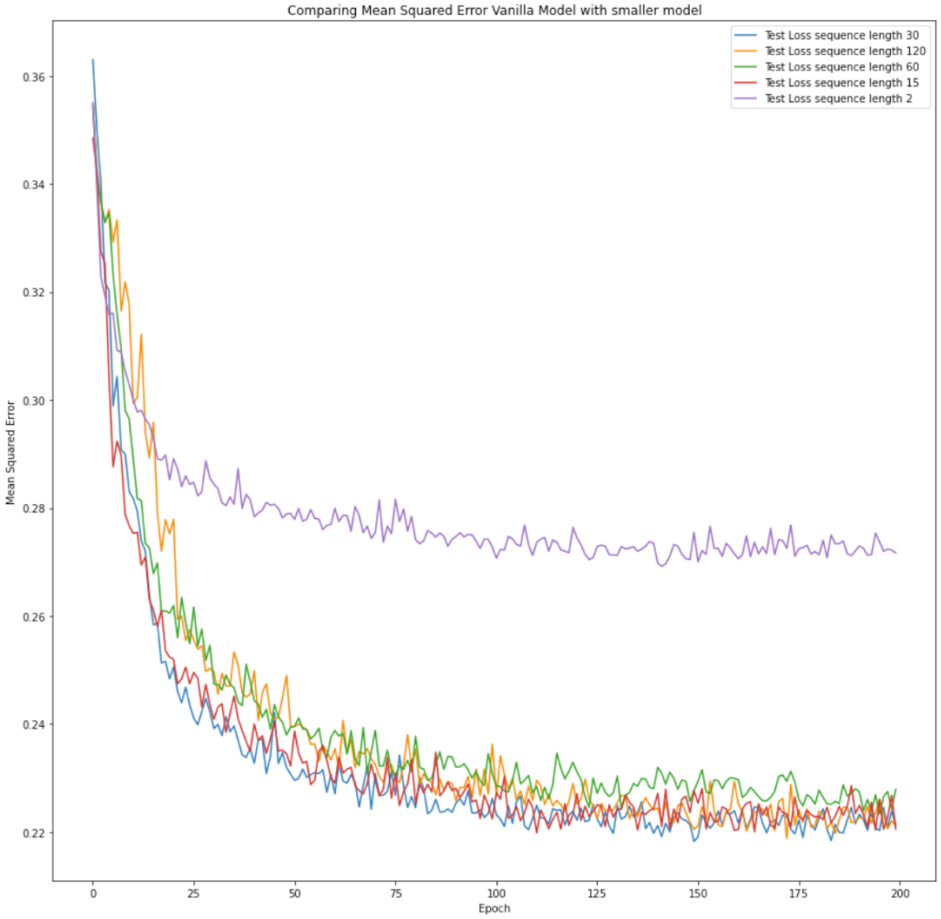



Figure 51: Plot Training results using different sequence length

The training results in the test set show that a sequence length between 15 and 30 is the best choice. Shorter sequences (2) are significantly worse and longer sequences (60, 120) although the error rate is similar, they are worst in absolute results and in time performance. In terms of time efficiency, a shorter sequence is the better choice. Further experiments would be needed to determine an optimal sequence length.

# 4 Conclusions

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

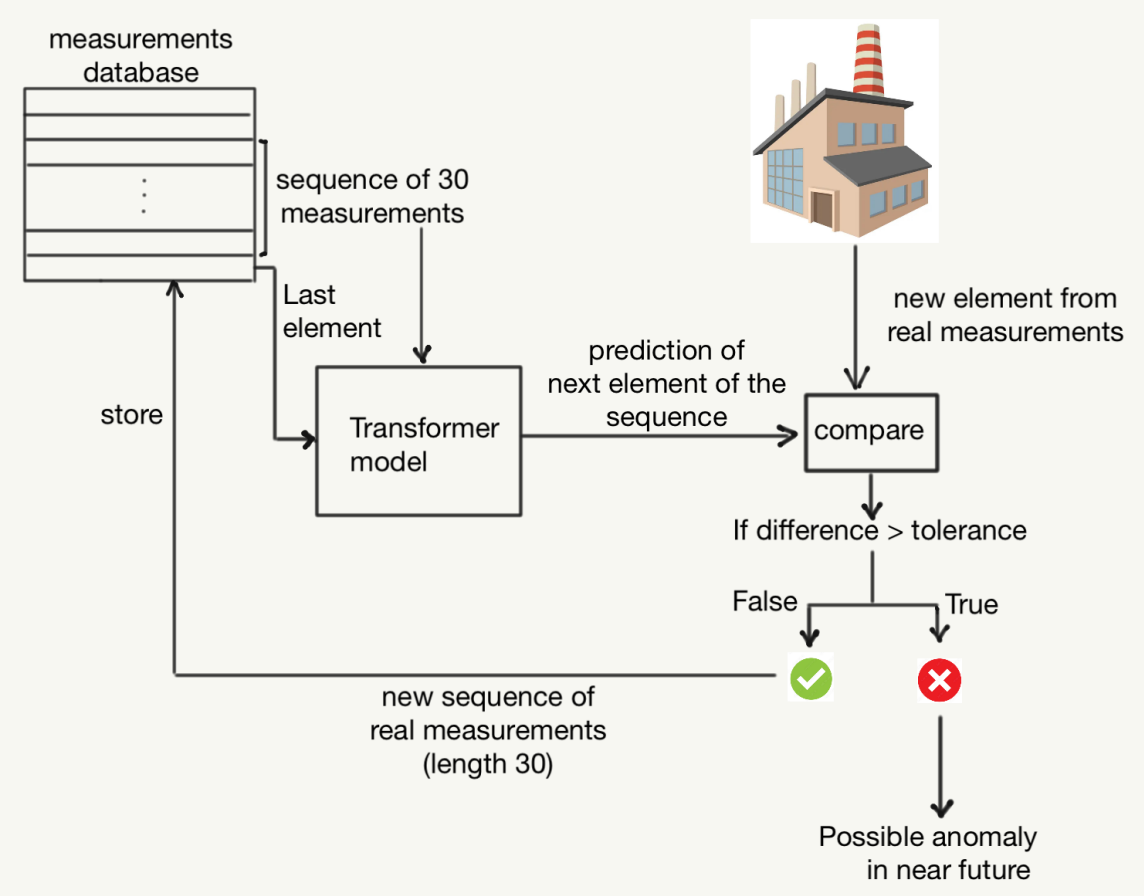



Figure 52: Schematic description of the entire project

After the first training with a kind of “vanilla”[69] model, the results in terms of Mean Squared Error over all samples and features of the sequence show a great improvement over the baseline models. Further improvements of this model showed that this error could be minimised even more, as we can see in this table:

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

| Algorithm parameters | MSE Training set | MSE Test set | Training Time (200 Epochs) | Training time per Epoch | Trainable parameters |
|---|---|---|---|---|---|
| Baseline model, output = input | 0.472 | 0.483 | - | - | - |
| Baseline model, FFN, 4 hidden layers 200 epochs | 0.272 | 0.288 | 22 minutes | 4 seconds | 237898 |
| Sequence length 30 | | | | | |
| 6 Encoder Layers, 1 Decoder Layer, 6 Heads. SGD. 200 Epochs | 0.242 | 0.258 | 6 hours 30 min. | 83 seconds | 542042 |
| 6 Encoder Layers, 1 Decoder Layer, 6 Heads. ADAM. 200 Epochs | 0.304 | 0.315 | 7 hours 30 min. | 100 seconds | 542042 |
| 6 Encoder Layers, 1 Decoder Layer, 6 Heads. SGD Mom. 200 Epochs | 0.192 | 0.221 | 8 hours 15 min. | 2 minutes | 542042 |
| 1 Encoder Layers, 1 Decoder Layer, 1 Heads. SGD Mom. 200 Epochs | 0.215 | 0.232 | 3 hours 15 min. | 38 seconds | 155872 |
| 10 Encoder Layers, 5 Decoder Layer, 9 Heads. SGD Mom. 200 Epochs | 0.193 | 0.218 | 14 hours | 3 minutes | 1165530 |
| Sequence length 120 | | | | | |
| 6 E. Layers, 1 D. Layer, 6 Heads. SGD with momentum. 200 Epochs | | | | | 542042 |
| Sequence length 60 | | | | | |
| 6 E. Layers, 1 D. Layer, 6 Heads. SGD with momentum. 200 Epochs | | | | | 542042 |
| Sequence length 15 | | | | | |
| 6 E. Layers, 1 D. Layer, 6 Heads. SGD with momentum. 200 Epochs | | | | | 542042 |
| Sequence length 2 | | | | | |
| 6 E. Layers, 1 D. Layer, 6 Heads. SGD with momentum. 200 Epochs | | | | | 542042 |

Figure 53: Complete Transformer Training Results

The best results in the test set are obtained with a transformer model with 10 Encoder Layers, 5 Decoder Layer, 9 Heads and using SGD with momentum. In terms of time efficiency, this was not the best choice, but as mentioned before, this trade-off between efficiency and results should be taken care in the future

Another aspect is the fact that there are very highly correlated features and that some of the features have a distribution that resembles a normal distribution, which could open up a possibility to improve the model when processing the data. Therefore, improvements could be made in the future for which we could use the previously calculated models as a new baseline model.

Recalling the original assumption: to be able to predict what values should be receive from the sensors to confirm correct functioning of the machines in the factory. At this stage I cannot prove the validity of this assumption. For that I would need more data and more experiments in the real scenario. For example, at the moment I don't know when these errors, although a big improvement over the baseline models, would be acceptable, or at the moment I don't know how big, and in which features, the difference between predicted and real values should be to confirm a malfunction in the factory machines. But I would like to remind the reader that the aim of this work was not to create a working model for exactly this use case, but to show whether using a transformer in a time series task is a good choice.

Theoretically, and considering the magnitude of the errors obtained in the experiments, I think this work is a good proof of concept for further research in this area.

# 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 , May. 5, 2022

[17] ‘torch.nn.TransformerEncoder’. https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoder.html , May. 5, 2022

[18] ‘torch.nn.Transformer’. https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html , May. 5, 2022

[19] ‘torch.nn.TransformerDecoder’. https://pytorch.org/docs/stable/generated/torch.nn.TransformerDecoder.html , May. 5, 2022

[20] ‘torch.nn.Linear’. https://pytorch.org/docs/stable/generated/torch.nn.Linear.html , May. 5, 2022

[21] https://www.simplypsychology.org/short-term-memory.html , May. 5, 2022

[22] https://dev.to/shambhavicodes/let-s-pay-some-attention-33d0 , May. 5, 2022

[23] https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/ , May. 5, 2022

[24] https://pytorch.org/docs/stable/generated/torch.optim.Adam.html , May. 5, 2022

[25] https://pytorch.org/docs/stable/generated/torch.optim.SGD.html , May. 5, 2022

[26] https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html , May. 5, 2022

[27] https://pytorch.org/docs/stable/generated/torch.nn.utils.clip_grad_norm_.html , May. 5, 2022

[28] https://machinelearningmastery.com/how-to-know-if-your-machine-learning-model-has-good-performance/ , May. 5, 2022

[29] https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-squared-error , May. 5, 2022

[30] https://yanlinc.medium.com/how-to-do-feature-selection-dimension-reduction-883c844aaaf6 , May. 5, 2022

# List of Figures

# List of Code Snippets

# 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 neighbour 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) |