FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Time Series Forecasting

Mikkel Godtfredsen
`xrq510@alumni.ku.dk`
`https://github.com/mikkelgod/Bachelors-Thesis`

14. June 2021

Supervisor: Stefan Horst Sommer

1 Abstract

Time series forecasting is an interesting subject in the field of machine learning as it combines the challenges of capturing the structure in long sequences with essentially ”predicting” the future. This project explores the *transformer*, an advanced model architecture implementing the concept of *attention*, and compares it to the more simple *Long Short Term Memory* model, a form of gated recurrent neural network that should be able to learn long term dependencies. This project focuses on evaluating and comparing the two models' abilities in making multi-step predictions which is where the challenges in time series prediction are most clear.
The project is concluded with the sentiment that further experimentation of the transformer architecture would be needed to pinpoint the specific contribution of the important elements of the transformer, this includes attention, positional encoding, and masking. However, based on the results of the experimentation and evaluation of the models, it becomes clear that the transformer model performed much better at predicting sequences of time series data than the LSTM model, and attention in the space of time series forecasting is definitely a concept worth exploring further.

Contents

2 Introduction

A time series is a sequence of data containing a temporal element, whether it be information about stocks, temperature measurements over a period, or the daily spread of a virus. In machine learning this is an interesting area of research as time series forecasting is a problem domain that consists of basically predicting the future based on historical data.
For a long time, based on time series data's sequential nature, the most widely used methods for time series forecasting have been recurrent methods such as recurrent neural networks and gated methods such as the long short term memory network. However, these types of methods still run into hurdles which make it hard for them to learn the long term dependencies that are key in predicting time series data. In an attempt to find new ways to tackle the challenge of ”predicting the future”, this project will try to examine the use of transformers and *attention* for time series forecasting, comparing them to the more mainstream methods. In this project, the focus will be on predicting multiple time steps as this is where both model architectures will be sufficiently challenged and should give a better visualization of how the models perform differently.

Each section will try to provide a theoretical background to each type of model, showing what makes them compatible for time series forecasting, and also showing the parts that make them different, and hopefully what differentiates their results from each other. From each type of model the results are presented and compared, and lastly their differences are evaluated and reflected on. To round of the project, a discussion on the method of approach is presented with the aim of giving the reader an idea of why different choices were made and why certain elements were left out.
This project is written with the mindset that the reader is knowledgeable on machine learning and data science, but is not necessarily an expert in the field of the ideas presented here, and on which this project hopefully should elaborate on.

3 Dataset

The dataset consists of time series data showing power consumption over 32 different European countries[1] shown in megawatts in hourly steps starting from the 1st of January 2015 and ending on the 30th of September 2020. This amounts to 50400 hours of time series data, which is roughly 5.7 years.
Below the whole dataset is shown, and a clear seasonality is seen in the data, with the most power consumption being around the winter seasons.

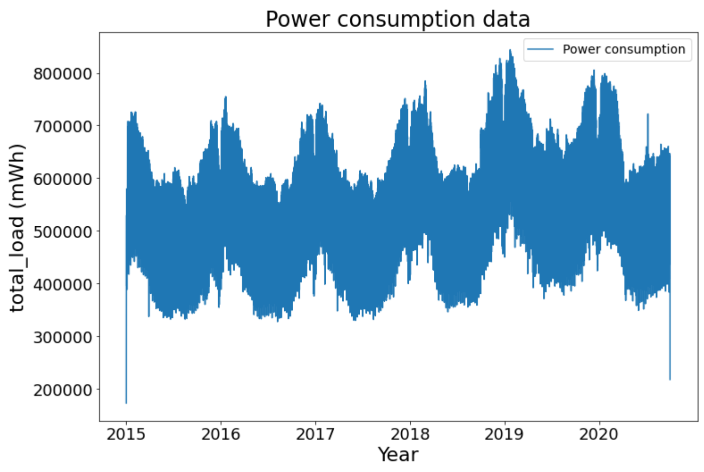

Figure 1: Plot of the whole dataset

[1]https://data.open-power-system-data.org/time_series/

For the purpose of evaluating different methods for time series forecasting, the most important use of this dataset did not lie in differentiating between power consumption from the different European countries or delve into the waters of multi-variate predictions. But rather to use the dataset as a whole as a stepping stone, and simplify the problem, so that the focus could be on evaluating the different models, and more easily compare them to each other. For this the specific features showing power consumption from the different countries were extracted and summed them up to have one feature, *total load*, thus turning the problem into a single feature problem as apposed to multi-variate prediction.

Furthermore, the data was subsampled, which means that the hourly data was turned into daily data. This serves the purpose of keeping the data at a decent size, not removing to many data points, while still being able to see some interesting dynamics from timestep to timestep. As shown below, the power consumption can be seen fluctuating on a weekly basis, while still keeping in line with the seasonality and overall trend of the whole dataset.

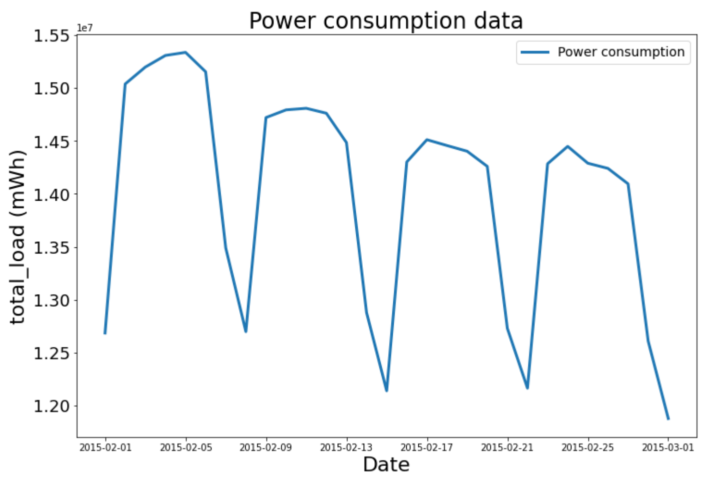

Figure 2: Approximately a month's worth of daily data

Throughout the project, the dataset is augmented into what is called a time series dataset. This method is general across different models, and is what makes it possible to train both the simple and advanced model, and to make not only single-step predictions but also multi-step prediction.
The method consists of making data sequences of sources and targets, which are sequences of time series data taken from the dataset, where the target sequence is shifted a number of steps, corresponding to the wanted prediction length.
In the case of the simple model, the source can be viewed as the training input and target is the wanted output which is what the actual output of the model is compared to. In the case of the advanced model, as it is an encoder-decoder architecture, the model needs both source and target for training.

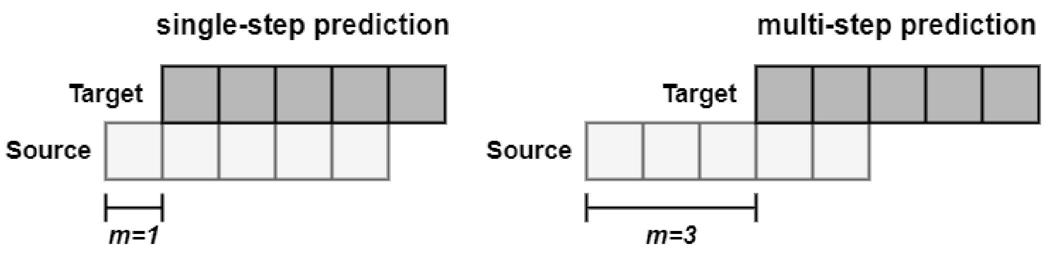

Figure 3: Visualization of data sequences for single-step and multi-step prediction

4 LSTM-RNN

This section will try to provide the necessary theory behind recurrent neural networks, focusing on the *Long Short Term Memory* model (LSTM). Such recurrent models are the generally the default methods which are chosen to work on sequential data such as in time series prediction. This explanation should hopefully give an idea as to how these types of models are used for predicting time series and why they are suitable for the problem in this project. Lastly, the results of the models performance on the power consumption data set are presented, showing the capabilities of the LSTM.

4.1 Theory

A recurrent neural network (RNN) is a model for processing sequences of data, and contrary to simpler feed forward neural networks where a sequence of data would be processed and have weights and features learned separately, a recurrent neural network can share weights across a sequence. Hereby lies the strength of an RNN as this parameter sharing can lead to generalization across sequences. An important feature of this type of model is that each member of the output is a result of the previous output, thereby giving the model its name [GBC16].
Below, figure 4 shows on the left a circuit diagram of the input entering state h and the recurrence with a time delay of a single step shown by the clock. On the right the circular diagram is unfolded showing each state at time step t as being a result of the previous state at $t-1$ as well as the input at step t.

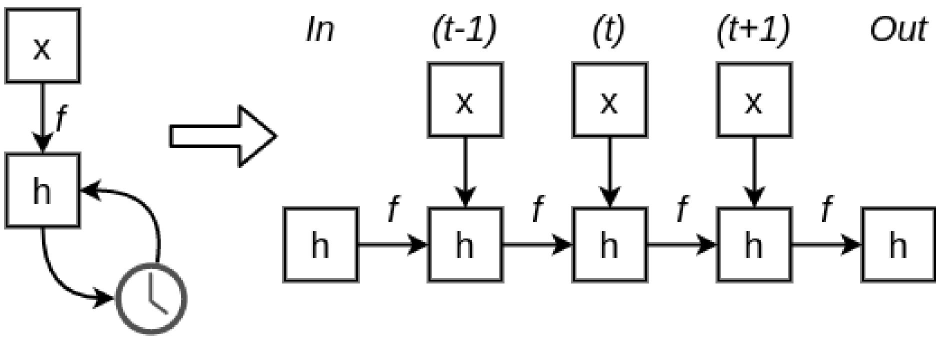

Figure 4: Unfolded recurrent network with no output

Like most neural networks, in between the input and the output layer lies the hidden layers. These consist of cells holding the weights that are given to the inputs and fed through the network. After each hidden layer, a function is applied which alters or shapes the output given to the next layer, this is an activation function. These building blocks can then be put together to form the neural network. The output of a hidden layer can be computed as follows:

$$h^{(t)} = \sigma(W_h x^{(t)} + U_h h^{(t-1)} + b_h)$$

where $h^{(t)}$ is the output as timestep t, σ is the activation function, W_h,U_h, and b_h are parameter matrices and vector. $x^{(t)}$ is the input as timestep t and $h^{(t-1)}$ is the hidden state from the previous timestep.

Backpropagation is used to calculate the gradient with respects to the loss function and the model parameters of a network. In the case of a recurrent network, the specific algorithm that is used is called *backpropagation through time*, as the model parameters are shared throughout the network and the output at a single timestep depends on the previous timesteps, and the gradients need to be calculated recursively [GBC16].
However, this recurrent nature can lead to a problem, which is very general when working with deeper neural networks, which is where one could start to observe vanishing or exploding gradients [GBC16]. A vanishing or exploding gradient occurs when training the model using backpropagation,

the model weights are updated by multiplying with a value corresponding to the partial derivative of the loss functions with respects to the weights in the current iteration. Now, for deep architectures, this means that very small or very big weights could potentially be multiplied with themselves many times, thus leading to weight values that either become so small or so big that they vanish or explode[GBC16].

One approach to solving this problem is using a form of RNN called a *gated network*, and more specifically the LSTM model.
The core of the LSTM lies in the introduction self-loops and gates confined within a LSTM cell, which helps to control the information flow [GBC16]. The self-loop is an internal recurrence within a cell and there is still the overall recurrent nature of the network.

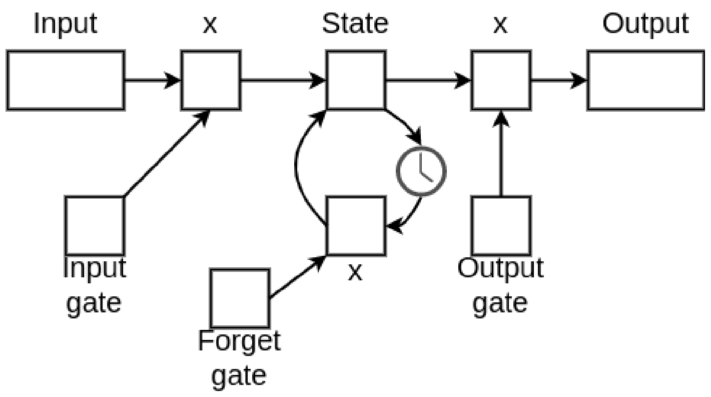

Figure 5: LSTM Cell

The external input gate, $g_i^{(t)}$, controls whether the value of the input itself is computed into the state through a *Sigmoid* activation function.

$$g_i^{(t)} = \sigma(b_i^g + \sum_j U_{i,j}x_j^{(t)} + \sum_j W_{i,j}h_j^{(t-1)})$$

Within the self-loop is the state unit which is controlled by the forget gate, $f_i^{(t)}$, also using a *Sigmoid* activation function, and it looks as follows:

$$f_i^{(t)} = \sigma(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)})$$

here x is the current input vector and h is the current hidden layer vector. b^f, U^f, and W^f are the biases, input weights, and recurrent weights for the forget gate respectively.
The state cell, $s_i^{(t)}$, is then computed, with the conditional self-loop, as:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)})$$

where b, U, and W are the biases, input weights, and recurrent weights from the input cell.
All the while, the output gate, $q_i^{(t)}$, can shut off the cells output completely, and is also using a *Sigmoid* activation function.

$$q_i^{(t)} = \sigma(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)})$$

again b^o, U^o, and W^o, being the biases, input weights, and recurrent weights for the output gate. Together, these functions make up for the output, $h_i^{(t)}$, of the whole LSTM cell:

$$h_i^{(t)} = \tanh(s_i^{(t)})q_i^{(t)}$$

In theory, this should make a recurrent neural network, in the form of a LSTM, suitable for time series forecasting. However, despite the fact that this type of model should be able to learn longer dependencies, it might struggle on predicting many steps ahead while keeping a high amount of precision.

4.2 Implementation

The LSTM model was implemented using `pytorch`, and in the simplest form of this type of model, with a single LSTM layer. For the other hyperparameters, which are the learning rate and the number of hidden layers, a grid search was run, finding the best combination of learning rates of $\{0.1, 0.01, 0.001\}$ and number of hidden layers $\{10, 50, 100, 200\}$. Generally, the best learning rate seemed to be 0.01, and the number of hidden layers fluctuated between 50 and 200.
For training the model, and evaluating it, the data was split into 60%/20%/20% training, validation, and testing respectively. And also done in batches of around 36.
In training the model, a form of adaptive stopping was run, which trained the model until no improvement in validation loss was seen for a patience amount of 40 epochs. This was set relatively high, hoping that the model would possibly improve over a longer period of time, also with the knowledge that this type of model usually runs a decent amount of epochs, however as will be seen below it did not make much of a difference.

4.3 LSTM Results

Below in this section, the results from different runs of the LSTM model can be seen, with different examples of multi-step predictions, showing the capabilities of the LSTM model. The first predictions are made with 50 days and predicting 25 days ahead, and the next ones are predicting 50 days ahead.

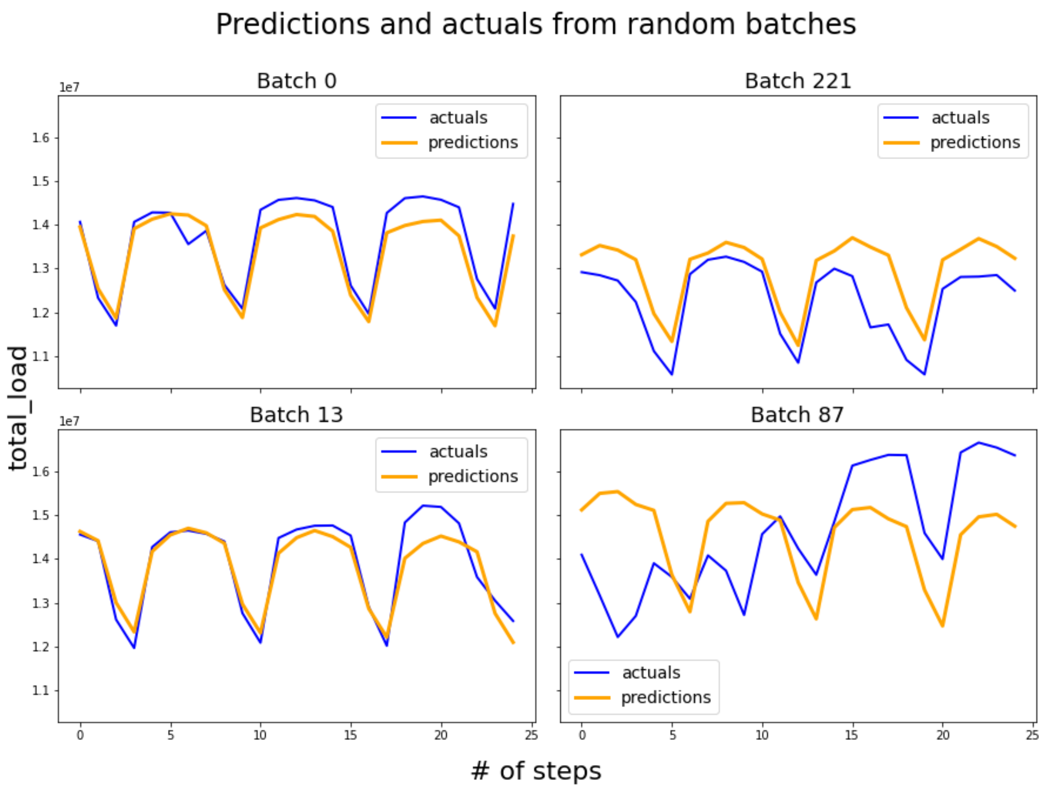

Figure 6: LSTM multi-step prediction of 25 days ahead, with look-back length of 50

Above is shown four different batches of the LSTM predictions and the corresponding actual

values. What is interesting to note, is that it looks like the LSTM network has made some pretty good predictions, in most cases, the orange line is right on top of the blue. However, one could speculate that the model does not deviate much from the standard, as most of the predictions follow the same structure. This means that the model in this case is simply predicting around the mean values of the part of the data on which the predictions are based on. In the cases where there are anomalies or deviations from the mean in the actual data, the LSTM prediction fails to follow. This is very much the case in batch 87 for example.
This sentiment is also mirrored in the loss, where it can be seen from the plot below that the validation loss does not converge at all and even shows signs of overfitting. It should also be noted that the training ran for a relatively low number of epochs, even despite the fact that patience was set to 40, so after around 20 epochs, validation loss did not improve further for 40 epochs. Giving an indication that this model would not be able to learn very well.

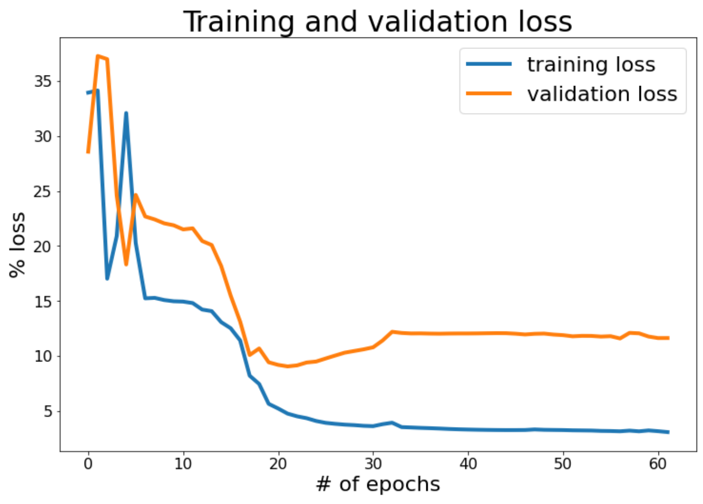

Figure 7:

The missings of the LSTM model can furthermore be backed up by increasing the number of steps on which the model makes predictions to 50 days, where the point is emphasized, especially with batch 94 and 170 seen below.

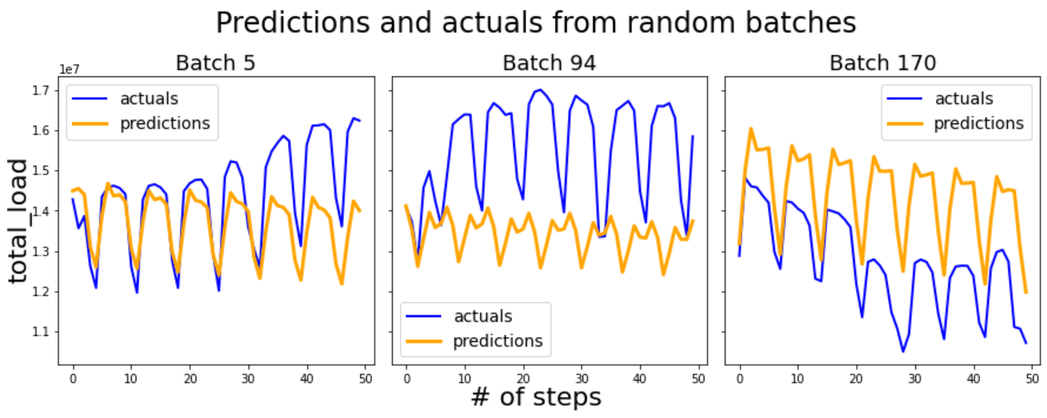

Figure 8: LSTM multi-step prediction of 50 days ahead, with look-back length of 50

Lastly, to have another more quantitative and more direct point of comparison, the average distance between the predicted values and actuals values across batches are presented. Here the above sentiments are visualized where it can be seen that the predictions on average get worse and worse the longer the prediction sequence gets.

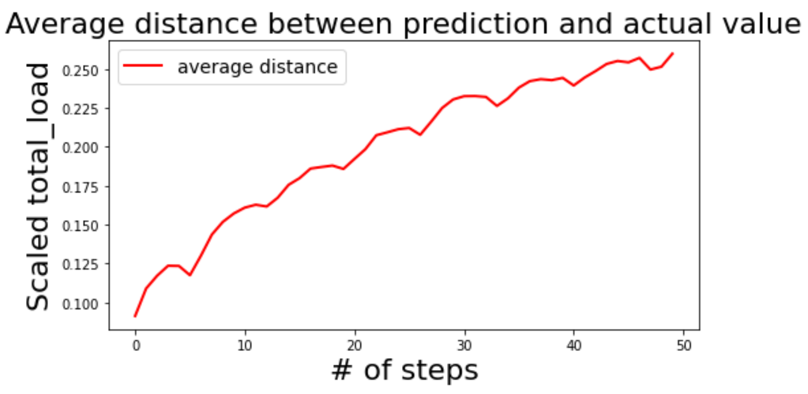

Figure 9: Average distance between predicted and actual values for the LSTM model

5 Transformers

Next up in this section, the theory behind the transformer is introduced, including the main features comprising mainly of the concept of attention, positional encoding, and masking, and how they work in the form of an encoder-decoder architecture. Furthermore, an explanation of how the transformer can be suited for time series forecasting will be presented, including the modifications needed to make it work. And lastly the non-trivial parts of the implementation will be shown followed by the results of the model's performance on the power consumption data set.

5.1 Theory

The transformer is an encoder-decoder architecture, much like an auto-encoder, where the input is encoding with certain information, changing the structure of the input, and then decoded to change it back to something similar to the original structure. Among other things, what makes the transformer special is that it utilizes what is called *attention*, which will be elaborated on below. A transformer can in theory be used to perform the same tasks as recurrent neural networks or convolutional neural networks, but by removing things like recurrent dependencies, it becomes possible to parallelize calculations. This together with attention, and adding elements such as positional encoding, masking among others, should set this type of model above others in these types of tasks, including time series forecasting, in both performance and precision [VSP+17].

5.1.1 Attention

Attention works by mapping a set of queries and pairs of key-values to an output. The output is calculated by a weighted sum of the values, where the weights are assigned by the likeness of the key and the corresponding query [VSP+17]. This score is calculated from the function:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

where Q, K, and V are the queries, keys, and values respectively, with d_k as the dimensions of the keys and queries, and values V of dimension d_v. In the case of the attention layer in both the encoder and decoder, the query, key, and value is simply the normalized input which is returned from the positional encoding.
Also a *Softmax* activation function is applied before multiplying with the values, which creates the weights by returning values between 0 and 1.
Now this above is what is called *scaled dot product attention*, which only calculates attention for a single head, or layer. An evolution of this is *multi-head attention*, and the difference here is that multi-head attention gathers more layers of the single-head attention layers, which in theory makes

it able to focus on several time steps at once. This is done by applying the single-head attention function to n different linear projections of the queries, keys, and values, which are concatenated, projected once again:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_n)W^O$$

where each head is:

$$head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

And $W_i^Q \in^{d_{model}\times d_k}, W_i^K \in^{d_{model}\times d_k}, W_i^V \in^{d_{model}\times d_v}$, and $W^O \in^{hd_v\times d_{model}}$ are parameter-matrices for the projections [VSP+17]. Both scaled dot-product attention and multi-head attention is visualized below.

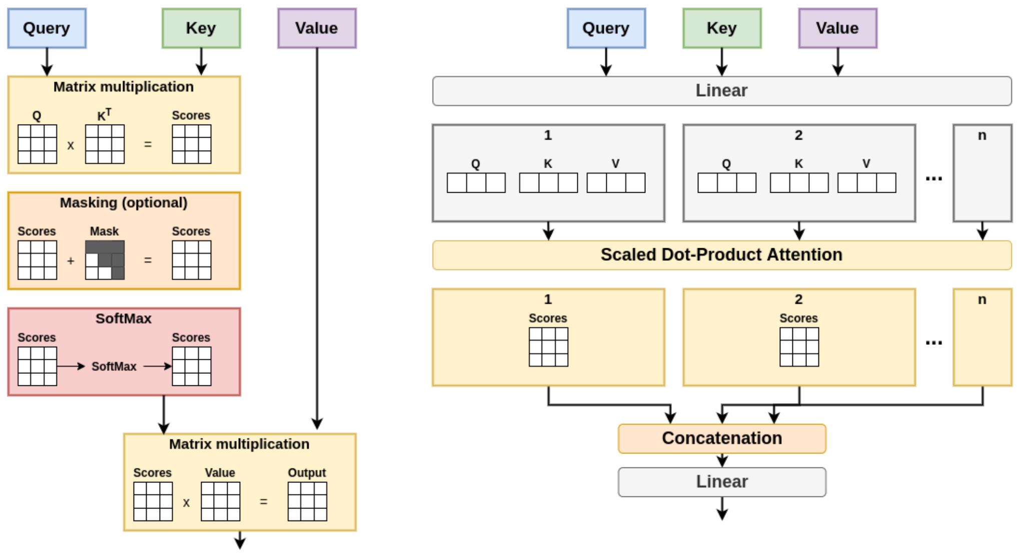

Figure 10: (Left) Scaled Dot-Product Attention (Right) Multi-Head Attention

Another piece of the transformer is added to make sure that the decoder cannot look ahead and share information about future parts of the sequence, this is called masking.
Masking is implemented by creating a square matrix tensor of the same size as the attention scores, with zeros below the diagonal and $-\infty$ on the diagonal and above. The matrix or mask filter, is then added to the scores. Essentially acting as a weight matrix that weighs the later parts of the sequence lower, thereby hiding them from the model, making sure that it cannot simply look ahead in the sequence to predict the next entry [LE18].

5.1.2 Positional encoding and embedding

Now the transformer model contains one more important element that differentiates it from other encoder-decoder models, which is essential as it injects information about the structure of the data sequence into the model. What is aptly called positional encoding is necessary as the encoder-decoder model does not contain the same recurrent structure such as an recurrent neural network or a convolutional neural network.
A token's position in a sequence is calculated and injected using the continuous functions *sine* and *cosine*, and given the position and dimension of the token:

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

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

With these functions, an encoding matrix is created with matching dimensions of the data, and then the two are added together to create the encoded data. An intuitive way to think about the encoding vector is as a number represented in binary, where each bit and their position gives the number. This is essentially the same case with this positional encoding, but instead of bits, floats are used, ranging in values between 2π and $10000 \cdot 2\pi$ on the sine and cosine wavelengths [Kaz19].
Normally, as the transformer from [VSP+17] was originally devised for semantics analysis, the embedding layer was tasked with converting the language data to vectors, but as this project concerns time series data, in this case it is a just a linear transformation that maps the data's feature dimensions to the dimensions of the model.

5.1.3 Normalization

The normalization layer was implemented by the following function:

$$\frac{\alpha_{d\_model} * (x - \mu_x)}{\sigma_x + \epsilon} + \beta$$

where α and β are vectors of size *d_model* containing ones and zeros respectively, and $\epsilon = 0.000001$, essentially computing the standard score by subtracting the mean of x from x and dividing by its standard deviation.

5.1.4 Feed Forward Neural Network

Each layer contains a simple feed forward neural network, which in essence are two linear transformations with a `ReLU` activation between, followed by dropout.

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

These are present to act as a sort of convolution, where the projections are the same, from 128 dimensions to 4×128, and back again, different layers contains different parameters[VSP+17].

With these elements, the structure of the transformer can now be visualized, and can be seen consisting of these different layers below.

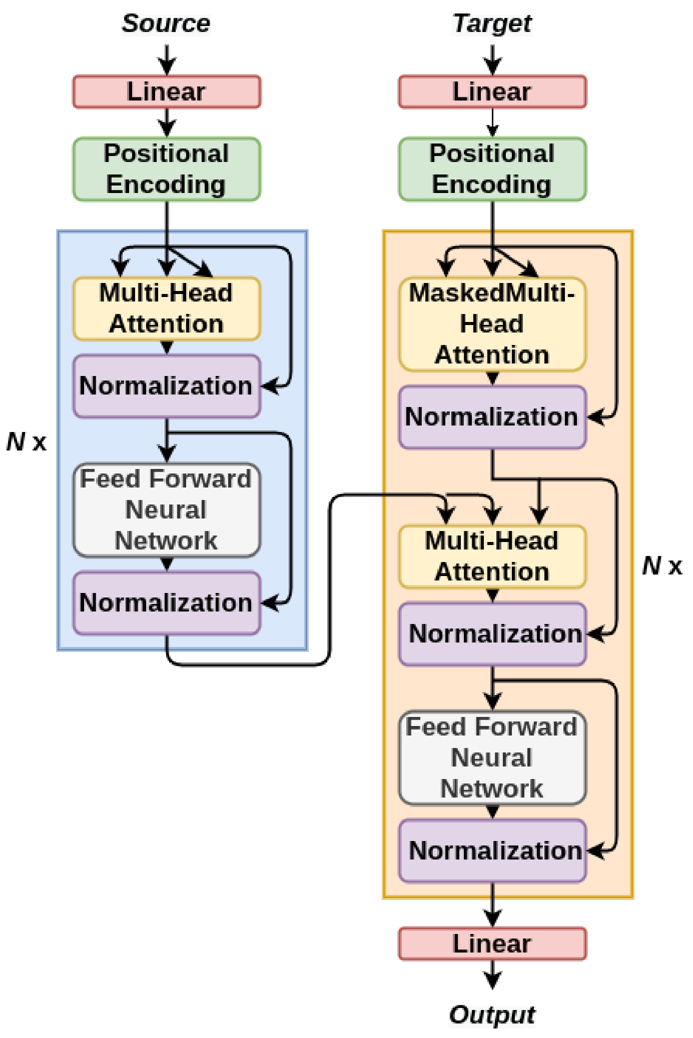

Figure 11: Transformer architecture

5.2 Implementation

For the implementation of the transformer, `pytorch`'s neural network library was used. The actual implementation of the different layers of the model simply follows all the layers presented above, instantiated in different classes to match their theoretic module counterpart.

As for the LSTM model, the time series dataset was split into 60%/20%/20% train/validation/testing. Furthermore, as the implementation uses `pytorch` dataloaders, the training was done in batch sizes of 36.
Furthermore, in terms of the actual hyperparameters of the transformer, following the recommendations of [VSP+17], the model was initialized with $d_{model} = 128$, 6 encoder/decoder layers, and 8 attention head layers. For the loss function, *Mean Squared Error* (MSE) was used to measure the average of squares of errors between the predictions and the actual values. The optimizer that was used was Adam, which in very simple terms can be seen as a combination of *RMSProb* and *AdaGrad* [KB17], together with a very low learning rate of 0.0001. In terms of learning rate, this implementation deviated from the article [VSP+17], as they used a dynamic learning rate, and instead a choice was made here to simply use a static low learning rate, as the size of the problem in this case was much smaller.
With regards to the number of epochs, in the article[VSP+17], they run a great number of epochs,

something that would not make sense here in terms of the size of the dataset and time spent, instead a form of adaptive stopping was run, where the validation loss was continuously evaluated, which was the case as for the training of the LSTM model, and which would stop training if the loss did not improve for a number of iterations, which was set to 10.

5.3 Transformer results

As mentioned, for the transformer, the focus was mainly on the multi-step predictions, as this is where the transformer in theory should separate itself from other ”simpler” methods.

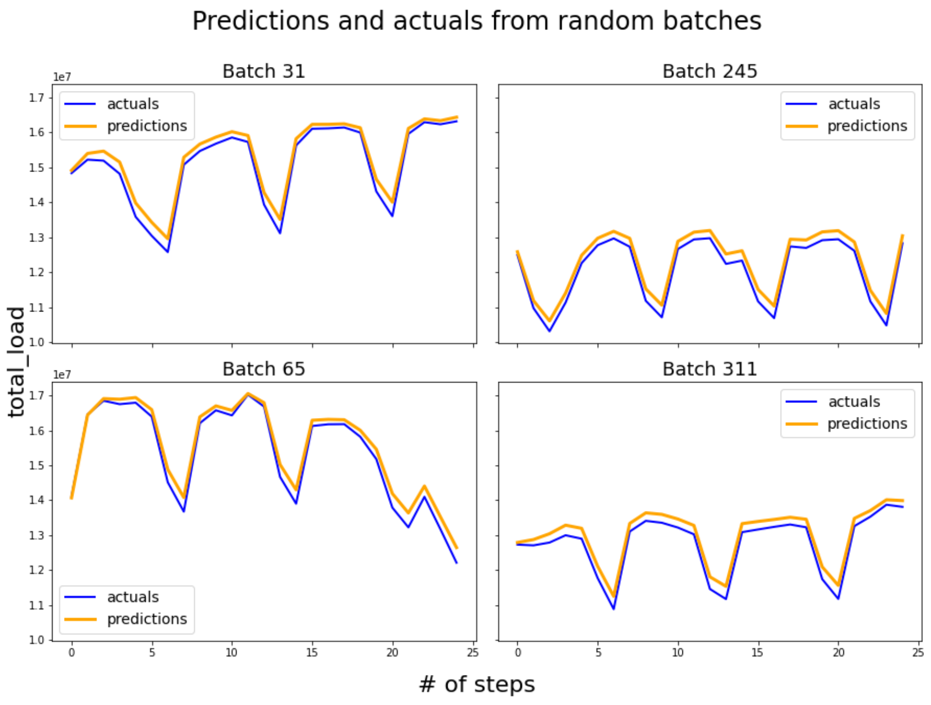

Figure 12: Transformer multi-step prediction of 25 days ahead, with look-back length of 50

What can clearly be seen from the different examples is that the transformer’s predictions lie close to the actual data from the test set. The small differences that can be noticed however is that there are small gaps between the predictions and actual values, where it looks like the predictions lie above the actuals values. These are small differences, and one could expect that they could be mitigated with further tuning of the hyperparameters, something that was not a focus of this project. Whether not doing this was the right approach will be discussed later in the project.
Furthermore, as the model was trained to make predictions 25 days ahead from the last 50 days, there is a decent amount of data to make predictions from, which could also be the reason for these relatively good predictions.
The classification of ”good predictions” is however not only based on visual inspection of the prediction plots, but also from the loss plot. What can be seen is that both the training loss and validation loss seem to converge towards almost zero loss after a relatively low number of epochs. It is interesting to note that the validation loss is lower than training loss, something that was generally the case throughout different runs.

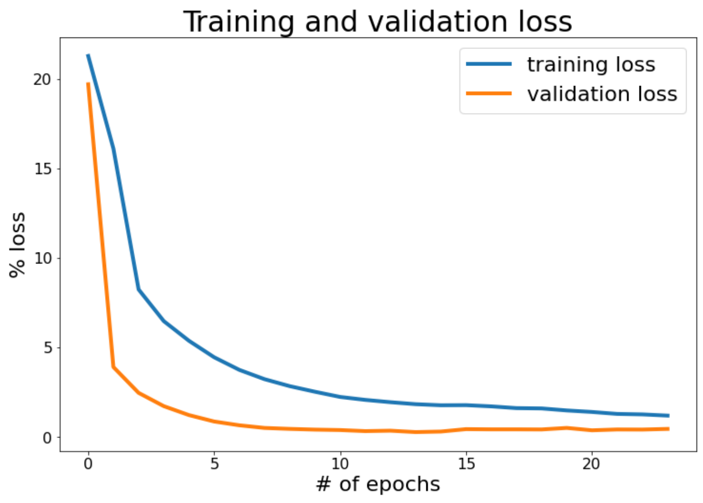

Figure 13: Training and validation loss for the transformer with 25 day predictions

Below, the capabilities of the transformer is emphasized by showing that the transformer does not need a very large amount of data to make its prediction with. As the plot shows, having the length of the look-back sequence the same size as the prediction still makes decent predictions, though the gaps between predicted and actual data become a bit larger.

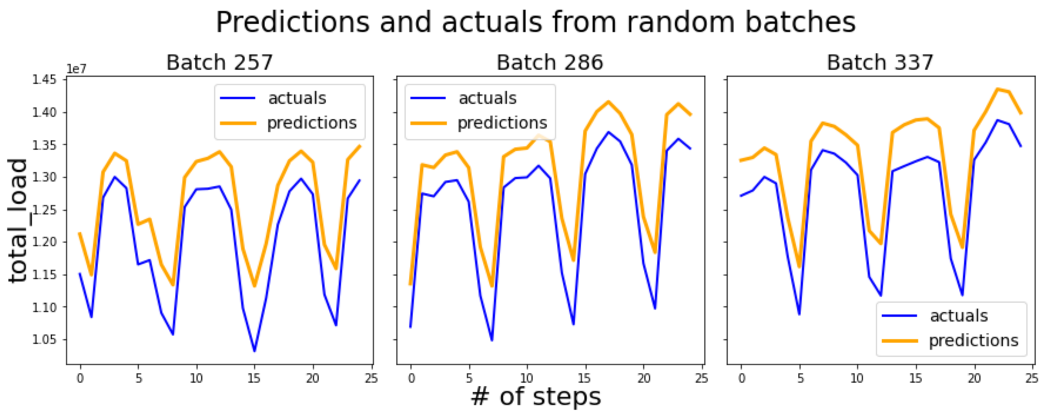

Figure 14: Transformer multi-step prediction of 25 days, with look-back length of 25

However, it does seem like the transformer is able to perform well even on longer predictions, as shown below, where really good predictions are shown on twice the number of steps, as long as it has enough data to make predictions from.

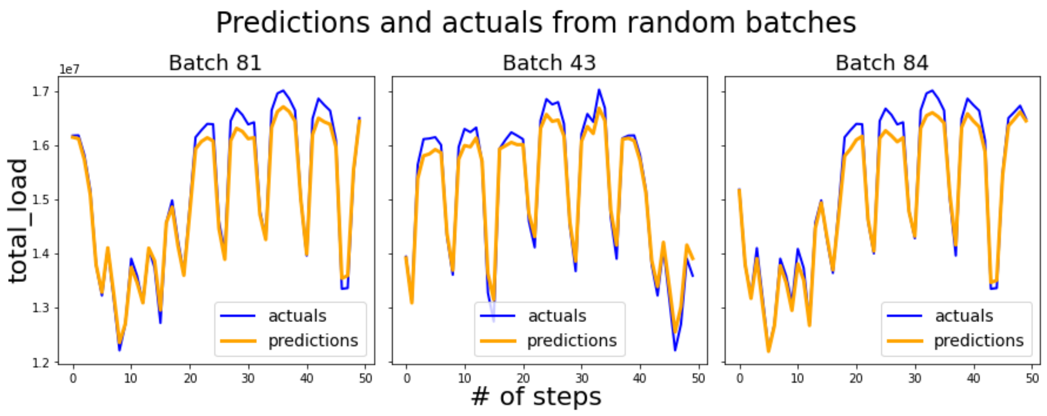

Figure 15: LSTM multi-step prediction of 50 days ahead, with look-back length of 50

Lastly, the average distance between the transformer predictions and actual values are plotted, in the same fashion as the LSTM model. This shows that the predictions are much more stable throughout the sequence, and though the difference in values are spiking, the predictions and actuals are much closer to each other overall, and the distance does not necessarily rise at the end of the prediction sequence.

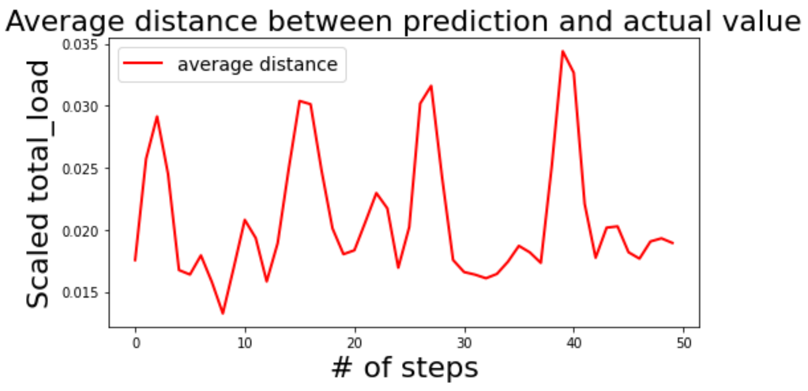

Figure 16: Average distance between predicted and actual values for the transformer

6 Evaluation

So far two different types of models and how they could be used for time series forecasting have been presented throughout in the report. Presented first was the relatively simpler LSTM, which builds on the idea of recurrency which is inherent in the nature of time series. Secondly, the transformer was presented, which is a model type that utilizes the concept of attention among other things, representing a more advanced type of model which can be augmented to make time series predictions, as the recurrent nature of time series is not inherent in the model architecture. In this section, an evaluation of the two model architectures is presented, focusing on the comparison of the results presented in the previous sections.

The first thing that is noticed is that the LSTM model had trouble making longer multi-step predictions. This was apparent as it output what seemed like good predictions, but what one could suspect were nothing better than simply the standard values from the sequence with which it based its predictions on. This was exemplified on figure 15 and 8, where the first figure showed "close" predictions, but where the second plot clearly showed great disparities between the actual values and the predictions. This was of course when the prediction length was increased from 25 days to 50,

posing a greater challenge. On the other hand, running the same experiment with the transformer, predicting the same days ahead based on the same number of days, much different results were output. On figure 12 and figure 14, it can easily be seen that the predictions in both cases were very close to the actuals values. Much closer than those of the LSTM model.
This sentiment was also mirrored in both of the models' loss that showed a much lower convergence point for the transformer, coming very close to zero loss for validation, and around $10 - 15\%$ for LSTM and in some experiments even worse, seen from figure 7 and figure 13. One strange thing that can be noticed however is that, generally throughout different experiments, the validation loss of the transformer seemed to be lower than the training loss, which should normally not be the case. One could suspect that this could be the cause of the validation set being only a part of the whole data set, and therefore containing only one set of seasonal fluctuation, in contrast to the training set that has around 3 years of seasonal fluctuations. And therefore by only validating based on this single year of data, the model had an easier time predicting on the validation data.
The last point of comparison is then the average distance between the predictions and actual values over each batch from both models. Here it was much more clear to see, that the transformer was a lot better at capturing the long term dependencies than the LSTM model and making multi-step predictions. From figure 9, it shows that the average distance gets greater and greater the longer the prediction sequence is, and in figure 16 it was easy to see that the distance stayed pretty consistently low throughout, even for the longer predictions.

When reviewing these results and speculating whether the complicated model is actually better than the simple one, there are some things that should be taken note of. Following the theory, and comparing the results from the transformer with those from the LSTM, one could be tempted to conclude that the transformer indeed is much better at time series forecasting. And furthermore, that the addition of the crucial elements such as masking, positional encoding, and last but not least, attention, is what made the difference. As without them, the transformer is not much more than a feed-forward neural network with some dropout. However, it can be difficult in practice to compare one type of neural network to another, especially when they vary so much in complexity, which is especially the case in this instance. But without testing the transformer model without its crucial elements, removing them bit by bit and trying different configurations, it is difficult to conclude whether they actually made a difference individually. Therefore, this is something to be very aware of, and is also the reason why it can not be simply concluded that the transformer with is simply the most favorable concept for time series forecasting. However, coming back to the actual results, based on what we are seeing, one could hypothesize that **something** about the transformer makes it a better model, and attention is a concept that is here to stay, especially when working on predicting time series data.

7 Discussion

Setting aside the clear differences between the two models and taking a step back to look at this whole problem in a broader perspective, there are some things that should be discussed. This section aims to provide some different views on the method of approach with handling the dataset, the different steps that were taken to make the implementation work throughout the whole work on the project, and also what possibly could be done to improve the model even further.

7.1 Independence of the data and features

On top of the reasons that were mentioned earlier with the absence of extensive testing of the model elements, another reason that a clear conclusion on the generalization capabilities of the transformer could not be reached, is the lack of independency of the validation and test data from the training data. Both of these come from the same signal as the training data which means that true generalization can not be reached as the model was essentially trained on the same data as it was evaluated

on, despite the fact that they come from different parts of the sequence. To counteract this, the model could have been trained on this dataset and evaluated on another dataset, or have been trained on data from some number of countries and testing on another countries data.
Despite this though, as mentioned earlier, a choice was made to dumb down and sub-sample the dataset to simplify the problem of time series forecasting. This was of course to make it easier to compare the bare bones model by making it a uni-variate prediction problem, but also in terms of simpler visualization in general. One could argue that this is generally not the right approach, removing much of the variation in the data, but as the focus of the project was on the models and their capabilities, and not the data, this felt like the right approach.
Though one could argue on the other hand that this removes some of the ability to compare models, seeing as multi-variate problems often represent how the world actually works. Other than that, this could also in some cases hurt the performance of the transformer, as having more features possibly could help the model make better predictions. An example of this is what is talked about in the paper [LALP20], which presents a library based on the transformer and attention architecture, specialized for time series forecasting. One of the novel technologies here is that extra features are added to the data, giving the model more information about known information from the future such as upcoming holiday dates among other things [LALP20]. This gives an indication on how the transformer possibly could thrive in more feature-heavy problems, something that contains more time series, more features, and more data. But using this library as another point of evaluating the performance of a transformer model is difficult, as this architecture not only utilizes both transformers and LSTM models, but also other technologies, making it even harder to pinpoint where the performance comes from.

7.2 Teacher forcing

Throughout working on the actual implementation of the transformer, the method for making predictions with the model went through different phases, before ending up with the final iteration. In first trying to adapt the transformer from [], from semantics analysis to numerical analysis, there were difficulties in making the transformer able to make predictions that were remotely close to the actual data. One of the methods that was tried out was *teacher forcing*, which is a method that consists of training the model to make single step predictions, and then compounding single step predictions to make multi-step predictions when evaluating on the test set [**?**]. To begin with, this made it possible for the transformer to make predictions that looked like the actual data, meaning that it could capture fluctuations on a daily and weekly basis, but varied widely in terms of the values, lagging steps behind or having values that were very different. However, in the same step of the implementation process where other crucial elements of the model were added, including masking, which were not added when trying teacher forcing, running the model without teacher forcing ended up working perfectly.

7.3 Hyperparameter tuning and experimentation

Following up on the statement that the transformer results possibly could have been improved by further hyperparameter tuning, it should be noted that this of course is the general case when working with neural networks. There were a few reason why this was not done, despite the fact that this is general practice when wanting to find an optimal model. Firstly, as mentioned earlier, the model on which the transformer in this assignment was based on had a set of recommended parameters, such as the number of encoder/decoder layers, number of attention heads et cetera. Though this in theory should not mean that the same model with these parameters would be compatible with the time series prediction problem in this project, especially when factoring in other parameters such as learning rate and different error measures. The combination of all these parameters make up a neural network model of a decent complexity, which leads to the other reason of the missing hyperparameter tuning. Though it is not an excuse to not implement and find the best hyperparameters, the amount of time it would take to test all the different combinations of parameters was an important factor in the decision that was made. Furthermore, keeping in line with the overall purpose of the project, tuning the transformer to find the perfect model would not necessarily add to the wanted purpose of this project, which was to simply compare the different models, and not to compare the perfect instances of said models.

Another point of critique of the approach of the experimentation of the models deals with the visualization of the results from the models. As mentioned earlier, in the case of the transformer, the loss plot showed a validation loss lower than its training loss, which generally would not make much sense. Setting aside the reason for this, to make this point even clearer, one could have compounded the loss plots from many different runs of the model. This is both the case for the transformer model, but also for the LSTM model. Showing an average plot of all these losses would of course give a more general idea of all the experiments, however again because of time constraints, and transformer loss plots that did not vary much in general, the choice was made to simply show a loss plot that seemed to represent the general behavior of the model.

8 Conclusion

Exploring the problem of predicting time series data, this project has shown what different methods of approaching the problem consist of and how they compare to each other. On one hand there was the LSTM model, representing the mainstream and more simple approach. In theory this type of model should be able to learn long term dependencies, and therefore should perform well in this domain of problems, however the LSTM model did not perform much better than simply predicting the mean values for multi-step predictions and got worse the longer the sequences it had to predict were. On the other hand there was the transformer, utilizing the concept of attention, making the model able to focus both on longer sequences and keeping the most important elements in view, while using masking to prevent the model from peeking ahead, and keeping the recurrent structure of the data with positional encoding. This all contributed to make predictions that were much closer to those of the actual data, and showing lower losses, than the LSTM did.
The question whether the transformer would generalize well however is still left up for debate as the utilization of the data as well as the constraints of the experiments prevent the elicitation of a clear conclusion.
And though it is impossible to specify the exact level of generalization and impact of each element of the transformer, this project is concluded with a partial confirmation of the idea that the transformer and attention definitely has a place in the space of time series forecasting.

References

[Ala18] Jay Alammar, *The illustrated transformer*, 2018, http://jalammar.github.io/illustrated-transformer/.

[EG20] Steven Elsworth and Stefan Güttel, *Time series forecasting using lstm networks: A symbolic approach*, 2020, arXiv:2003.05672.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, MIT Press, 2016.

[Kaz19] Amirhossein Kazemnejad, *Transformer architecture: The positional encoding*, 2019, https://kazemnejad.com/blog/transformer_architecture_positional_encoding/.

[KB17] Diederik P. Kingma and Jimmy Ba, *Adam: A method for stochastic optimization*, 2017, arXiv:1412.6980.

[LALP20] Bryan Lim, Sercan O. Arik, Nicolas Loeff, and Tomas Pfister, *Temporal fusion transformers for interpretable multi-horizon time series forecasting*, 2020, arXiv:1912.09363.

[LE18] Samuel Lynn-Evans, *How to code the transformer in pytorch*, 2018, https://towardsdatascience.com/\how-to-code-the-transformer-in-pytorch-24db27c8f9ec.

[VSP$^+$17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin, *Attention is all you need*, 2017, arXiv:1706.03762.

[WGBO20] Neo Wu, Bradley Green, Xue Ben, and Shawn O'Banion, *Deep transformer models for time series forecasting: The influenza prevalence case*, 2020, arXiv:2001.08317.