jamk

# Comparative analysis of GRU, LSTM, RNN, TCN, and WaveNet for hourly electrical consumption prediction

Abass Adeyinka
Juho Kuokkanen
Iana Daniel
Paladon Soothi-o-soth
Petri Vähäkainu

Neural Network Group exercise
12 / 2024
AIDA

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

**Description**

**Authors: Abass Adeyinka, Juho Kuokkanen, Iana Daniel, Paladon Soothi-o-soth, Petri Vähäkainu**

**Comparative analysis of GRU, LSTM, RNN, TCN, and WaveNet for hourly electrical consumption prediction.**

Jyväskylä: JAMK University of Applied Sciences, December 2024, 79 pages.

Artificial Intelligence and Data analytics, Master of Engineering g. master's thesis.

Permission for open access publication: Yes/No

Language of publication: English

**Abstract:**

The accuracy of predictions regarding hourly electrical consumption is essential for optimizing energy management and ensuring grid stability when faced with increasing demands. This study evaluates the performance of different neural network models such as Gated recurrent units (GRU), Long short-term memory networks (LSTM), Temporal convolutional networks (TCN, Recurrent neural networks (RNN), and WaveNet on this predictive task. Using the mansion dataset provided as the historical hourly electrical consumption, each model is trained and tested with identical data preprocessing for a fair comparison. Performance metrics used include mean absolute error (MAE), root mean square error (RMSE), and R squared ($R^2$) are used as accuracy indicators. The results highlight each model's strength and weakness in capturing temporal dependencies.

**Keywords/tags (subjects)**

See Project Reporting Instructions, Section 4.1.2 https://oppimateriaalit.jamk.fi/raportointiohje/4-opinnaytetyon-rakenne/4-1-opinnaytetyon-alkuosa/4-1-2-kuvailulehti/ - avainsanat

**Miscellaneous (Confidential information)**

For example, the confidentiality marking of the thesis appendix. See Project Reporting Instructions, Section 4.1.2.

**Figures**

**Tables**

# Introduction

In this group exercise, we study different neural networks that can be used to predict the electricity consumption of real property located in Jyväskylä, Finland. The types of neural networks investigated in this exercise are designed for predicting from time series data, such as energy consumption.

This first chapter summarizes basic information about the data we are using in the exercise and some insights from the data analysis phase.

Chapter 2 briefly introduces the technical background of RNN, LSTM, GRU, TCN and WaveNet model types. Each student has made a deeper study and implementation for one model type. Chapter 3 presents model-specific implementations and the results that have been achieved using these models.

The last chapter summarizes the prediction results from each model and provides the final answer to the following questions, as well as our opinion on the best model to predict the answers:

- Predict the next hour's electricity consumption
- Predict the next day's (24h) electricity consumption.

## 1.1 Basic information about the apartment

The main features of the property for which energy consumption will be predicted are listed below:

- Type: Detached house
- Type of heating: electric boiler, water pipes inside the floor.
- Area: Apartment 138m2 + storage 10m2 + garage 30m2 (semi-heated)
- Family: 2 adults (Mika is the teacher), 3 children (all in school)
- Electricity contract: Fixer price contract
- Air source heat pump (cooling / heating): from 2022
- Number of plug-in vehicles: 0
- Number of full electric vehicles: 0
- Other things: Woodstove, fireplace and sauna

As there is no electric car and the sauna is the only demanding consumer, we don't see anything from the above list that could be utilized in the model. As we see in the EDA and in the real predictions, the sauna brings some randomness to the consumption peak, but this will be discussed later in this exercise.

## 1.2 Feature engineering

From the original data, we used only the features Consumption and AirTemperature (°C). All the wind speed and other weather-related columns have been dropped as unnecessary.

We have created our own new feature using the hour information (0…24). We used sine and cosine functions to indicate the time, replacing other complex time functions and their encodings for the model. Then, we created additional features using the month information (1…12). We simply one-hot encoded the months and these are used then as features.

During the contact weekend when the group work started, we discussed and brainstormed various topics related to additional features and information that might help increase the model's accuracy, as listed below. However, we decided to keep the model as simple as possible because there are already many aspects that need to be investigated during the development process, such as the hyperparameters of different models.

- One discussion topic was the weekday, and it could be one-hot encoded as the feature, but in one quick testing it was not increasing the accuracy, and it was left out in the further studies.
- Covid 19: How to consider that Mika and his family have stayed longer at home during lockdowns. Should we make our own feature for that (0 or 1).
- Should we make more general attributes like at home (0 or 1) which can be used also for Covid-19 lockdown and all weekends.
- School holidays: Mika and his 3 children. Is consumption higher during the holiday season? Should we add week numbers to the data and use them to mark which weeks are school holidays? We already investigated past dates regarding school holidays in Jyväskylä, but we did not implement these in the final solution.

- **2019-2020:**
- School starts 8.8.2019
- Autumn break 14.–18.10.2019

- Christmas holiday 21.12.–6.1.2020
- Spring break 24.–28.2.2020
- Summer holiday: 30.5.2020

- **2020-2021:**
- School starts 12.8.2020
- Autumn break 12.–16.10.2020
- Christmas holiday 21.12.–6.1.2021
- Spring break 1.–5.3.2021
- Summer holiday: 5.6.2021

We have created a lag of 24 different datapoints/hours for Consumption (value_lag) and Temperature (temp_lag). We believe that these will help to accurately capture the changes in variables. So, we have the lag of one hour, two hours, and up to one day in their own columns.

## 1.3 EDA

As stated earlier, our approach from the beginning was to keep the exercise and the model as simple as possible. We created the correlation heatmap shown below, which indicates how different existing features correlate with each other. The highest correlation is between energy consumption and temperature, as electricity is used to keep the apartment warm, especially in the winter.

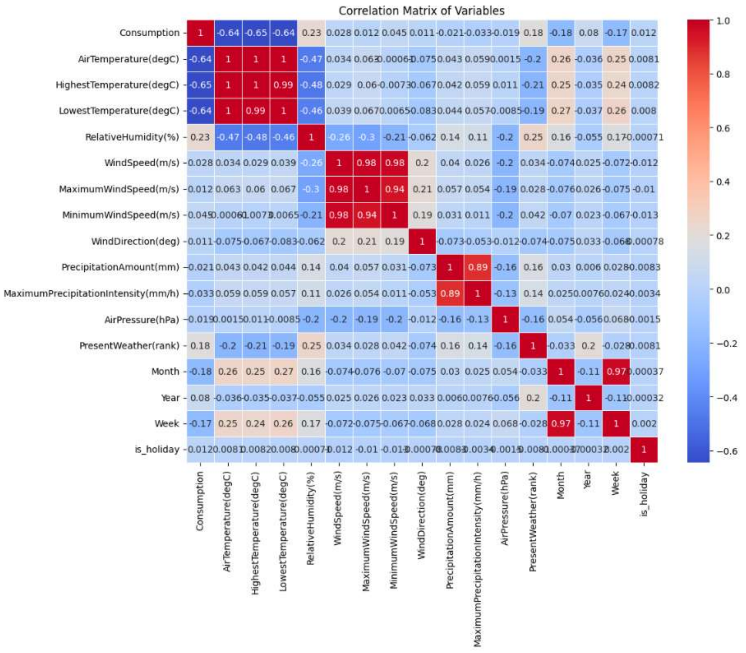



Figure 1. Correlation matrix of variables.

As can be seen from this correlation plot, there are three variables that have a high absolute positive correlation with the 'consumption' variable. 'LowestTemperature,' 'HighestTemperature,' and 'AirTemperature' correlate very strongly with each other. Therefore, including all three of these variables in the model to predict the target variable might cause a multicollinearity issue.

### 1.3.1 Hourly average electricity consumption

Based on the entire historical dataset, on average, energy consumption is high between 17:00 and 19:00.

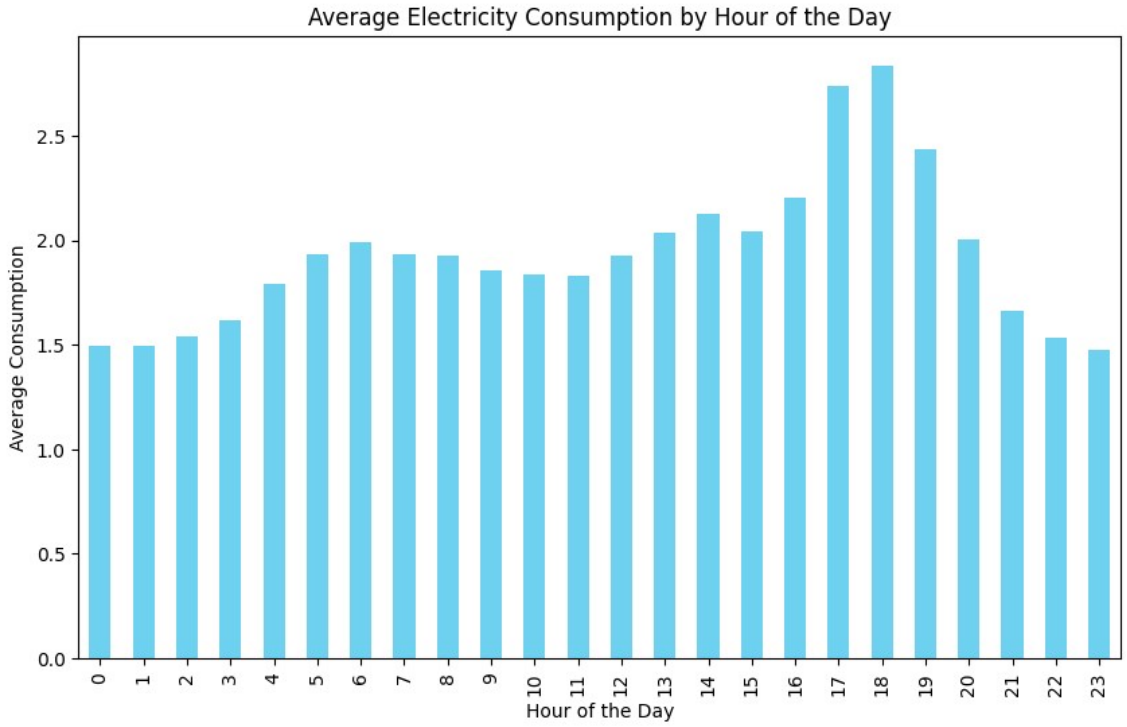



Figure 2. Average electricity consupmtion by hour of the day.

The graph below shows February 2018 as an example of hourly energy consumption changes throughout the day.

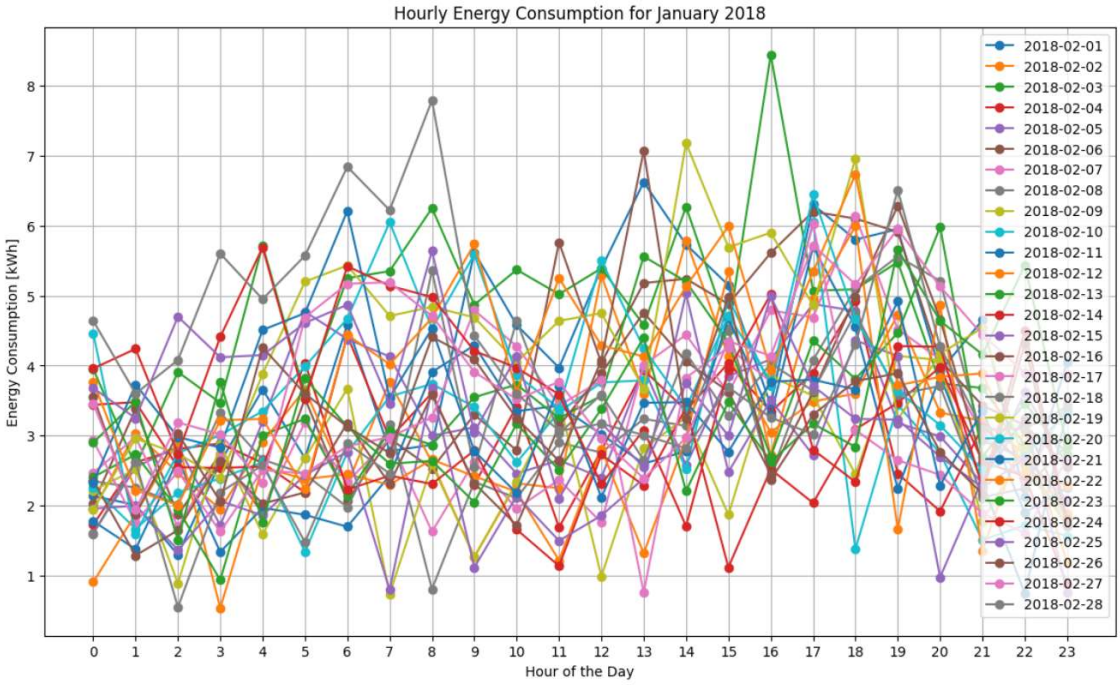



Figure 3. Hourly energy consumption for january 2018.

### 1.3.2 Distribution of consumption

Distribution of consumption is shown in the graph below. We can see that it is right - skewed since most of the values are clustered around a lower range. As we can see the mean value of consumption is 1.93 and it is higher than the median value of 1.77. Therefore, it suggests the presence of outliers in the data or extremely high usage periods. The standard deviation is 1.35 and it means that the variability in energy consumption is most likely influenced by other factors, such as seasonality, day of the week, or other external weather conditions.

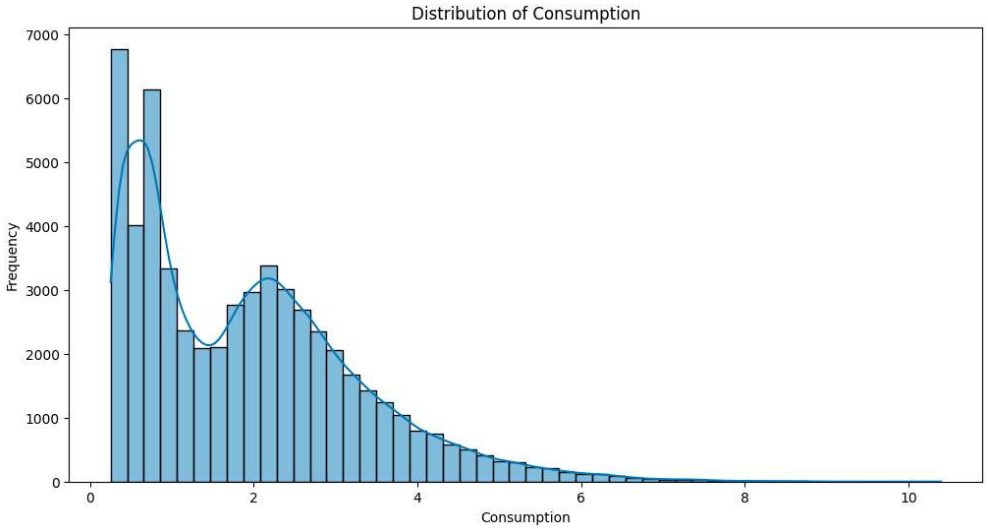



Figure 4. Distribution of consumption.

### 1.3.3 Seasonal overview

As we can see from the consumption by season graph, there are differences in the median consumption between seasons. For instance, there is a higher median consumption in winter compared to summer. This is due to increased energy needs for heating during cold months. The IQR range of consumption is also larger during winter, indicating a greater variability in energy use, likely due to fluctuations in temperature. There are outliers in each season, especially during winter and fall, showing certain spikes in energy consumption. These spikes are most likely due to the sauna, or the sauna combined with very cold weather, which significantly increases energy consumption. Overall, we can see that seasonality affects energy consumption.

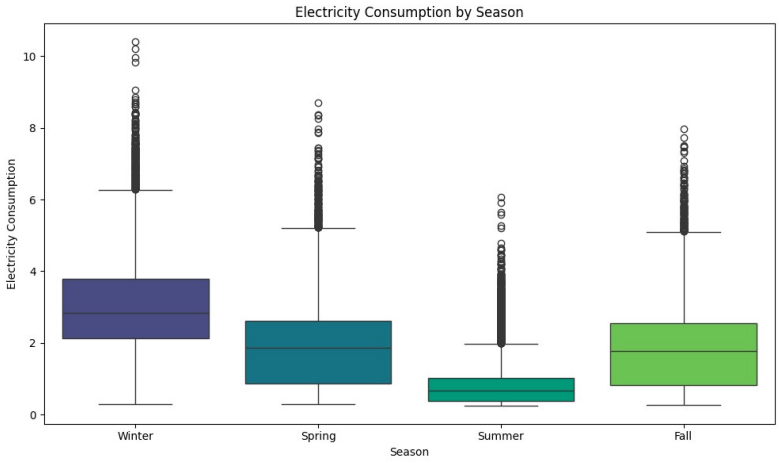



Figure 5. Electricity consumption by season.

To detect the outliers in the target variable, we used the 90th percentile to find the threshold for identifying rows with unusually high energy consumption. As can be seen from the graph below, the highest number of observations with the highest energy consumption were in January, February, and December. Again, this confirms that winter months are a significant driver of increased energy usage.

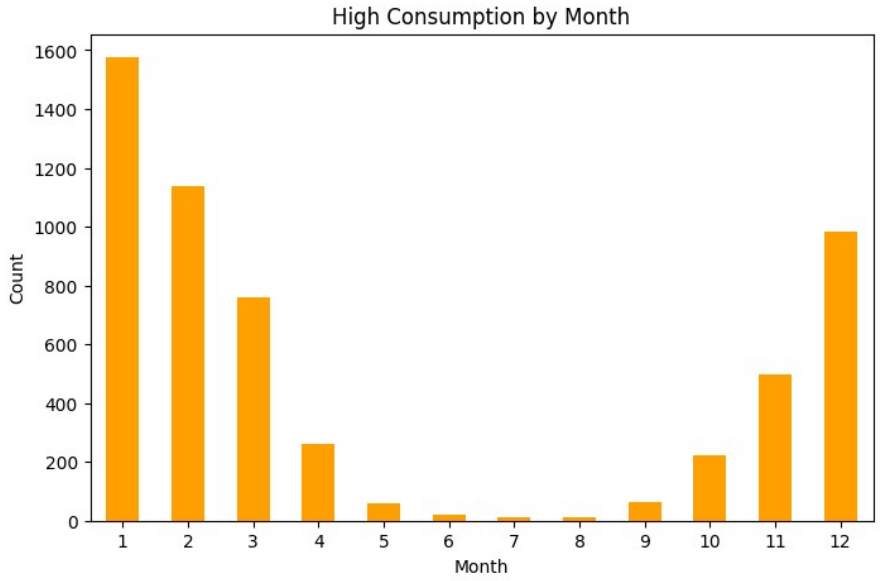



Figure 6. High comsumption by month.

### 1.3.4 Monthly average electricity consumption

The plot below shows that energy consumption has been quite stable over the different months and years. For example, there are no new loads, such as an electric vehicle, that might increase consumption.

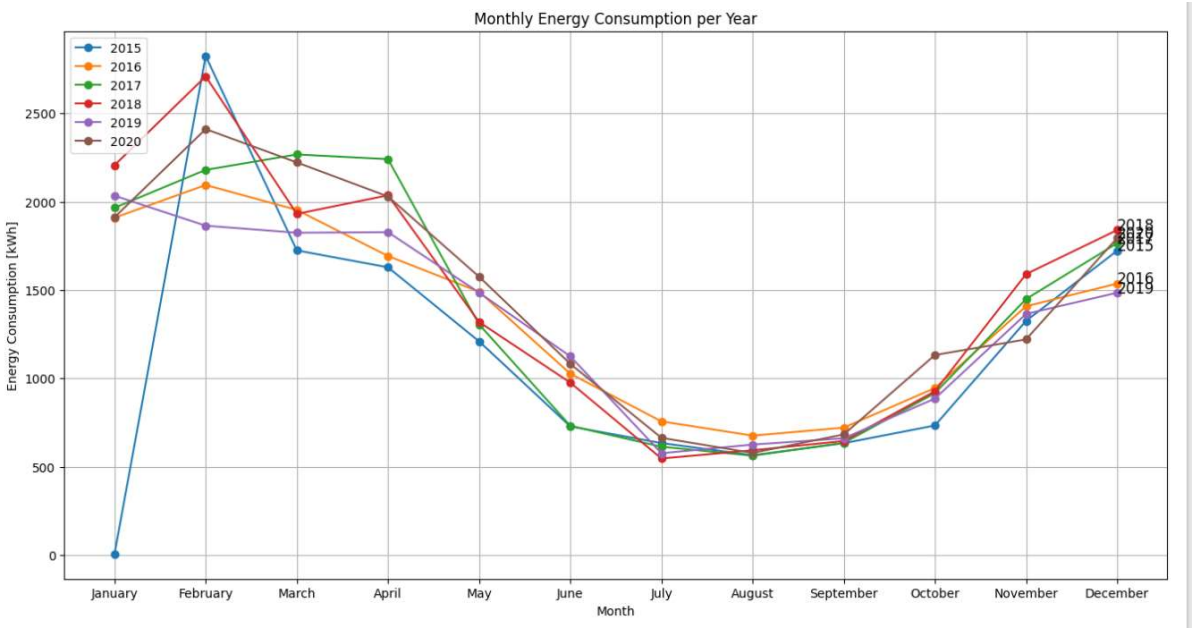



Figure 7. Montly energy consumption per year.

### 1.3.5 Weekly average electricity consumption

The plot below shows the same thing but in a weekly overview."

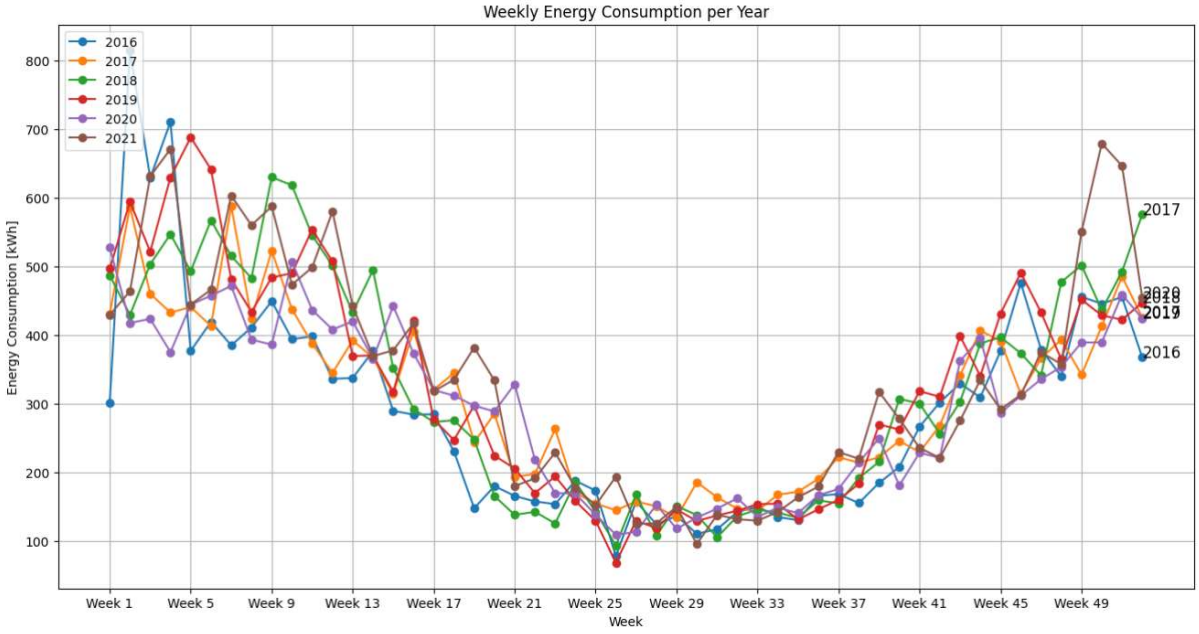



Figure 8. Weekly energy consumption per year.

## 1.4 Outliers / Missing values

The given dataset contains several missing values in the following columns (but only the first one is really used in building the model):

- AirTemperature(degC)
- HighestTemperature(degC)
- LowestTemperature(degC)
- RelativeHumidity(%)
- WindSpeed(m/s)
- MaximumWindSpeed(m/s)
- MinimumWindSpeed(m/s)
- WindDirection(deg)
- PrecipitationAmount(mm)
- MaximumPrecipitationIntensity(mm/h)
- AirPressure(hPa)
- PresentWeather(rank).

Most likely, the data source for all those listed parameters is the same, and for some reason, the weather data does not contain all data points. To tackle this challenge, we filled the missing values using a rolling window median for each column independently. The median value is less affected by outliers compared to the mean; therefore, the median value was used to fill the missing values. For rows where a rolling window median was not defined due to prior missing values, we used forward fill to propagate the last known value forward and backward fill to fill any remaining gaps from known values ahead for the “PresentWeather(rank)” column.

| | Consumption | AirTemperature(degC) | HighestTemperature(degC) | LowestTemperature(degC) | RelativeHumidity(%) | WindSpeed(m/s) |
|---|---|---|---|---|---|---|
| count | 56231.0 | 56231.0 | 56231.0 | 56231.0 | 56231.0 | 56231.0 |
| mean | 1.93 | 4.26 | 4.75 | 3.77 | 80.86 | 2.76 |
| std | 1.36 | 9.88 | 9.98 | 9.84 | 19.73 | 1.69 |
| min | 0.25 | -31.8 | -31.5 | -60.0 | 13.0 | 0.2 |
| 25% | 0.73 | -1.8 | -1.5 | -2.2 | 71.0 | 1.5 |
| 50% | 1.77 | 3.2 | 3.6 | 2.8 | 88.0 | 2.4 |
| 75% | 2.73 | 11.7 | 12.3 | 11.18 | 96.0 | 3.6 |
| max | 10.4 | 32.5 | 39.9 | 32.1 | 100.0 | 15.2 |

Figure 9. Overview of columns.

| MaximumWindSpeed(m/s) | MinimumWindSpeed(m/s) | WindDirection(deg) | PrecipitationAmount(mm) | MaximumPrecipitationIntensity(mm/h) | AirPressure(hPa) | PresentWeather(rank) |
|---|---|---|---|---|---|---|
| 56231.0 | 56231.0 | 56231.0 | 56231.0 | 56231.0 | 56231.0 | 56231.0 |
| 3.39 | 2.16 | 209.23 | 0.07 | 0.21 | 1011.36 | 9.55 |
| 1.9 | 1.53 | 92.24 | 0.35 | 1.16 | 12.25 | 23.87 |
| 0.3 | 0.1 | 1.0 | 0.0 | 0.0 | 957.7 | 0.0 |
| 2.0 | 1.0 | 142.0 | 0.0 | 0.0 | 1003.9 | 0.0 |
| 3.0 | 1.8 | 203.0 | 0.0 | 0.0 | 1011.8 | 0.0 |
| 4.3 | 2.9 | 296.0 | 0.0 | 0.0 | 1019.4 | 0.0 |
| 16.5 | 13.0 | 360.0 | 15.1 | 80.7 | 1053.2 | 87.0 |

Figure 10. Overview of columns.

The graph below shows energy consumption and three other variables that highly correlate with it. These variables are as follows: air temperature, highest temperature and lowest temperature over time. As can be seen from this plot, there is a

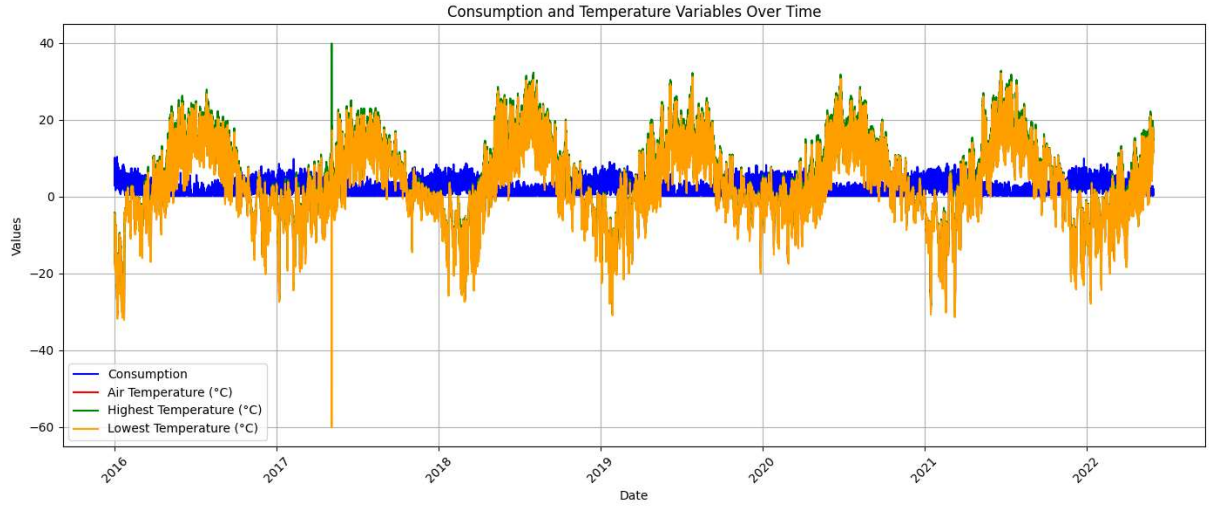



Figure 11. Consumption and temperature variables over time.

## 1.5 Base load of the house

The base load of the apartment can be investigated using the graphs below, which show the energy consumption from July 2018. The apartment was most likely empty for a long period, starting from 23.07.2018. There are no significant changes in energy consumption during that time. Normal base consumption is used for air ventilation, the fridge and the freezer. Sometimes these appliances are running, and sometimes they are off (due to the compressors).

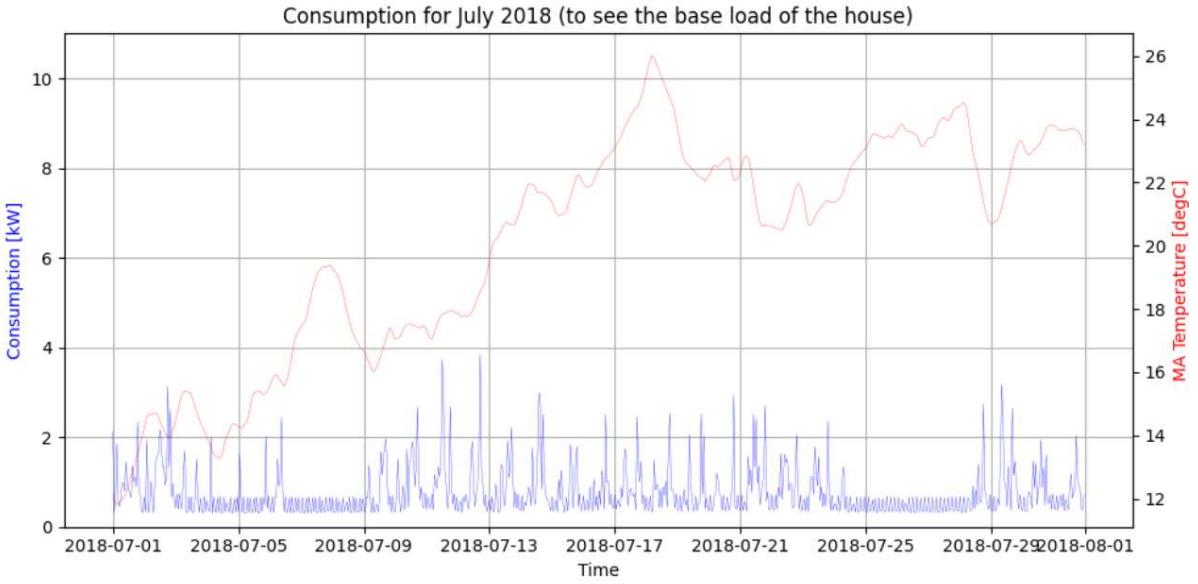



Figure 12. Consumption for July 2018.

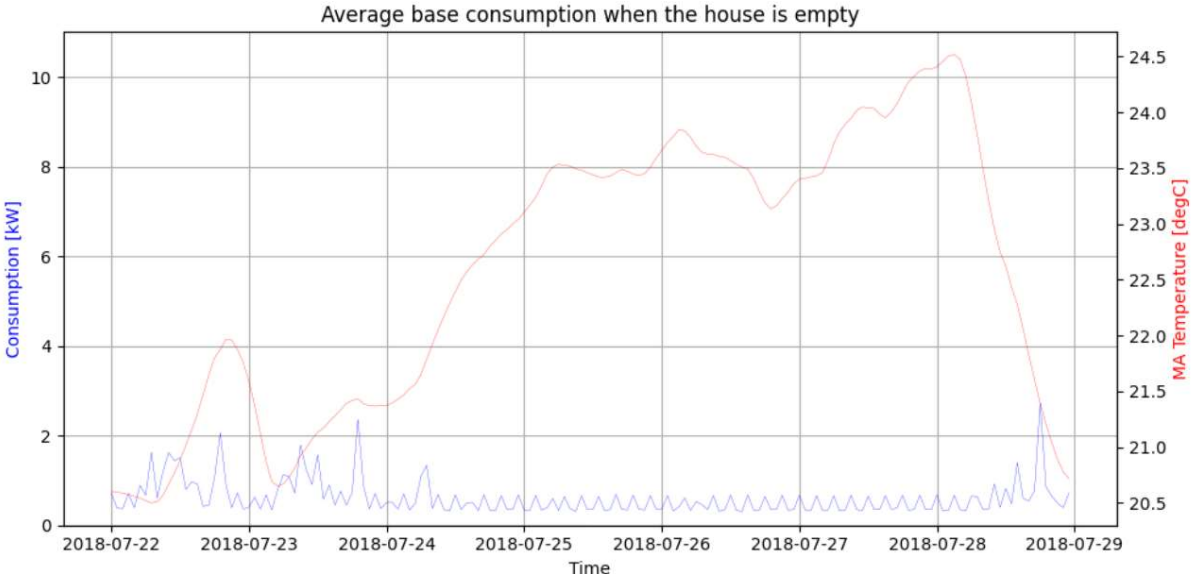



Figure 13. Average base consumption when the house is empty.

## 1.6 Sauna

Many findings from EDA and actual energy consumption predictions indicate that there are consumption peaks which are very challenging to capture and predict. Most likely, these peaks are due to Finnish sauna culture. Saunas significantly increase energy consumption, as their power can range from 3 to 9 kW.

A quick analysis of the data shows that there are significant energy peaks randomly occurring between 16:00 and 19:00. Most often, these peaks occur on Saturdays, but they vary on other days.

In the study below, situations when energy consumption increases by over 3.5 kW from the previous hour and then decreases by 2 kW in the following hour were filtered out. In the dataset, it happens as many times as listed below:

- at 10 - 3 times
- at 11 - 1 time
- at 12 - 8 times
- at 13 – 3 times
- at 14 - 11 times
- at 15 - 5 times
- at 16 - 19 times
- at 17 - 43 times
- at 18 - 65 times
- at 19 - 10 times
- at 20 - 3 times.

# 2 Introduction and technical background of Neural Network

## 2.1 RNN

**Recurrent Neural Networks (RNNs)** are a unique type of neural network architecture designed to analyse and recognize patterns in sequential data. This can include data like handwriting, DNA sequences, textual information, or time-series data, which is often encountered in industrial applications such as financial markets or sensor outputs (Goodfellow, Bengio, & Courville, 2016). RNNs can also process images by dividing them into sequential patches and treating these as a temporal sequence (LeCun, Bengio, & Hinton, 2015). They are widely used in tasks such as language modelling, text generation, speech recognition, image description creation, and video labelling (Sutskever, Vinyals, & Le, 2014).

Instead of transmitting information in a linear, one-way manner as in Feedforward Neural Networks (FNNs) or Multi-Layer Perceptrons (MLPs), RNNs use loops that allow the network to send information back to itself (Goodfellow, Bengio, & Courville, 2016). This looping capability enables RNNs to incorporate prior inputs ($X_0 \; to \; X_{t-1}$) along with the current input ($X_t$), thereby enhancing the ability to handle sequential data and capture temporal dependencies, which traditional Feedforward Networks struggle to process (LeCun, Bengio, & Hinton, 2015).

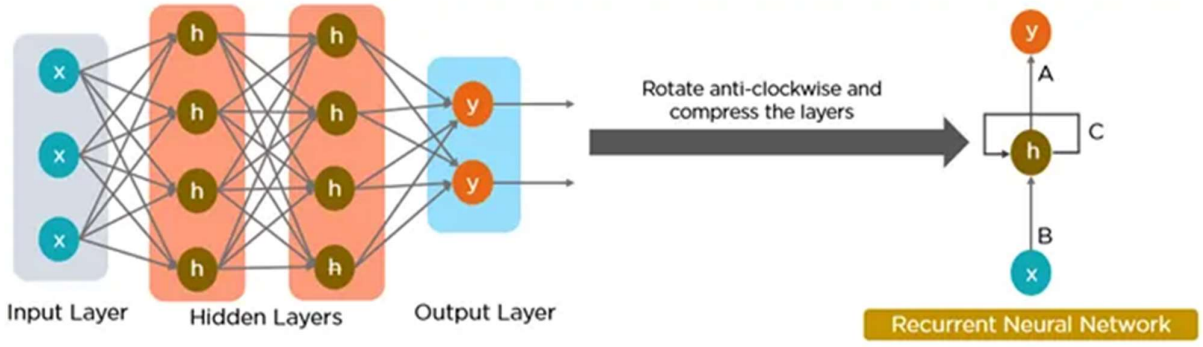



Figure 14. Architecture of the Recurrent Neural Network. (Kalita, 2024)

As presented in fig 1., RNN consists of the input layer, hidden layer(s), activation function, output layer, and recurrent connection. According to Kalita, 2024), RNN consists of the following:

- **Input Layer:** This is where the sequence data enters the network. For instance, in the context of a sentence, the input layer could receive the first word as a vector representation.

- **Hidden Layer:** The central component of the RNN, the hidden layer, consists of interconnected neurons. Each neuron processes the current input alongside the state from the previous hidden layer. This state acts as the network's memory, helping it interpret the current input in relation to past data.
- **Activation Function:** This function introduces non-linearity to the model, which is crucial for learning complex patterns. It modifies the combined input—comprising the current input and the previous hidden state—before forwarding it through the network.
- **Output Layer:** Using the processed information, the output layer provides the network's prediction. For example, in a language model, it may predict the next word in the sequence.
- **Recurrent Connection:** A defining feature of RNNs, this connection enables the hidden layer to pass its state (or memory) forward to the next step. It is analogous to a relay race baton, transmitting information about previous inputs to subsequent steps.

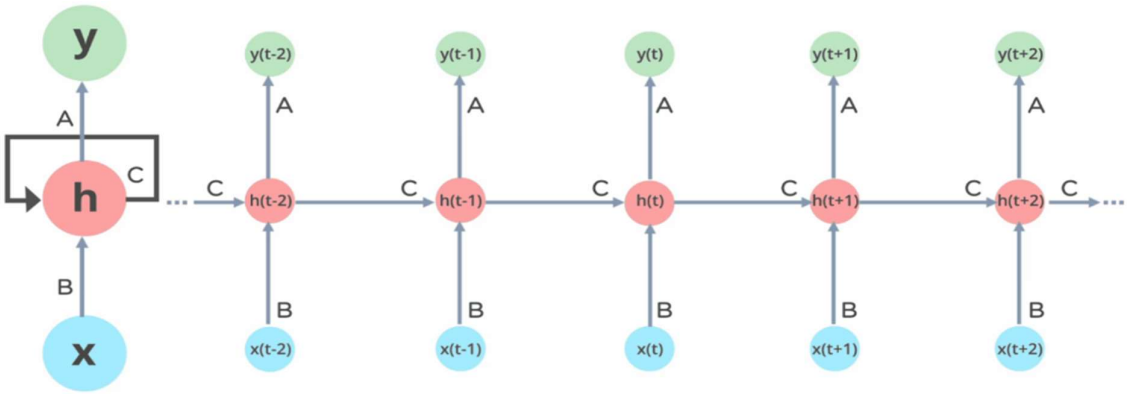



Figure 15. Information flow of RNN from input X to the output Y. (Kalita, 2024)

The architecture includes the input layer, hidden layer, output layer, and a recurrent connection. The hidden layer processes both the current input and memory from past inputs, allowing the network to interpret data in context, making it especially useful for sequential data analysis (Goodfellow, Bengio, & Courville, 2016). However, traditional RNNs face challenges like vanishing gradients, which limit their ability to capture long-term dependencies (Hochreiter & Schmidhuber, 1997). This issue is addressed by advanced architecture such as LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Units), which use specialized gates to regulate the flow of information and preserve long-term dependencies (Sutskever, Vinyals, & Le, 2014). These improvements allow RNNs to maintain performance over longer sequences, enabling them to handle complex tasks such as language modeling, machine translation, and speech recognition (LeCun, Bengio, & Hinton, 2015).

## 2.2 LSTM

LSTM stands for Long Short-Term Memory, and it is the special type of Recurrent Neural Network (RNN). It has a more complex structure than traditional networks. LSTM has additional memory cells and gates that allow it to selectively remember or forget information from time series. The internal memory is used to learn the sequences of inputs, which is why it would be the good choice for time series predictions.

One commonly used technique for anomaly detection in the time series is using LSTM autoencoders. The autoencoder is a type of neural network designed to copy its input to its output through an architecture consisting of an encoder and a decoder. The encoder compresses the input into a lower-dimensional representation, known as the bottleneck, which captures the essential features of the data. The decoder then reconstructs the input from this compressed representation. By training the autoencoder on normal data, the model learns to reconstruct this data accurately, and any significant deviation in reconstruction error can indicate an anomaly in the time series. This method leverages the power of LSTM networks to capture temporal dependencies, making it particularly effective for sequential data analysis (Hochreiter & Schmidhuber, 1997; Sutskever, Vinyals, & Le, 2014).

The LSTM model has three binary gates and memory cells as described below:

- Input gate: This controls the information flow from the current input and the previous hidden state to the memory cell.
- Forget gate: This gate controls the information flow from the previous memory cell to the current memory cell. This helps LSTM to selectively forget or remember information from previous steps.
- Memory cell: The internal state of the LSTM. Information is stored here but can be selectively modified by the input and forget gates.
- Output gate: This controls the information flow from the memory cell to the current hidden state and output.

The overview of LTSM network is shown in the figure below:

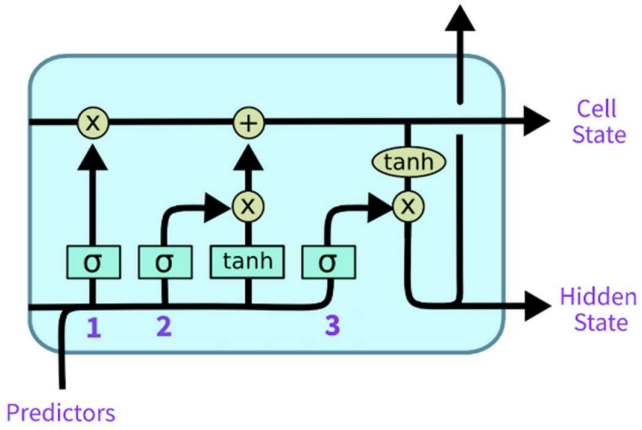



Figure 16. Overview of LSTM network.

In the figure above, the gates (σ) numbered 1 to 3 control the flow of information from the previous cell to the predictors. After determining the amount of relevant information to retain, the gates pass this to the tanh functions, which update the weights of the nodes. This process outputs two pieces of information: the cell state (long-term memory) and the hidden state (short-term memory) (Hochreiter & Schmidhuber, 1997). The gating mechanism is crucial for the LSTM's ability to retain important information across steps while discarding irrelevant data. This architecture enables LSTMs to effectively handle tasks like language modelling and sequence prediction, where maintaining context over long sequences is vital (Sutskever, Vinyals, & Le, 2014).

Building on the gating mechanism, the LSTM network's ability to selectively retain and discard information allows it to capture long-term dependencies effectively. By controlling the flow of data through these gates, LSTMs avoid issues like the vanishing gradient problem that traditional RNNs face, enabling them to maintain stable learning over long sequences (Hochreiter & Schmidhuber, 1997). This makes LSTMs particularly suitable for applications requiring the preservation of contextual information, such as speech recognition or machine translation (Sutskever, Vinyals, & Le, 2014). Moreover, the adaptability of LSTMs makes them robust for a wide range of sequential data tasks, ensuring accurate predictions despite complex input patterns.

## 2.3 GRUs

Gated Recurrent Units (GRUs), introduced by Cho et al. in 2014, are a type of recurrent neural network (RNN) designed to model sequential data by selectively remembering or forgetting information. With a simpler architecture than Long Short-Term Memory (LSTM) networks, GRUs are computationally efficient while maintaining performance, making them ideal for tasks like time-series forecasting, natural language processing, and speech recognition (Cho et al., 2014; Bahdanau et al., 2015). In time-series forecasting, such as household energy consumption, GRUs capture the relationship between past and future values. The model is trained on historical data, with Mean Squared Error (MSE) as the loss function, and performance is evaluated using metrics like Root Mean Squared Error (RMSE) and R-squared ($R^2$) (Kundu & Nasipuri, 2017; Zhou & Yang, 2015).

The GRU architecture consists of the following components:

- Input Layer: Takes in sequential data, such as a sequence of words or a time series of values and feeds it into the GRU.
- Hidden Layer: The recurrent computation occurs here. At each time step, the hidden state is updated based on the current input and the previous hidden state. The hidden state represents the network's "memory" of previous inputs.
- Reset Gate: Determines how much of the previous hidden state to forget. It takes as input the previous hidden state and the current input, producing a vector of numbers between 0 and 1 that controls the degree to which the previous hidden state is "reset" at the current time step.
- Update Gate: Determines how much of the candidate activation vector to incorporate into the new hidden state. It takes as input the previous hidden state and the current input, producing a vector of numbers between 0 and 1 that controls the degree to which the candidate activation vector is incorporated into the new hidden state.
- Candidate Activation Vector: A modified version of the previous hidden state that is "reset" by the reset gate and combined with the current input. It is computed using a tanh activation function that squashes its output between -1 and 1.
- Output Layer: Takes the final hidden state as input and produces the network's output. This could be a single number, a sequence of numbers, or a probability distribution over classes, depending on the task at hand.

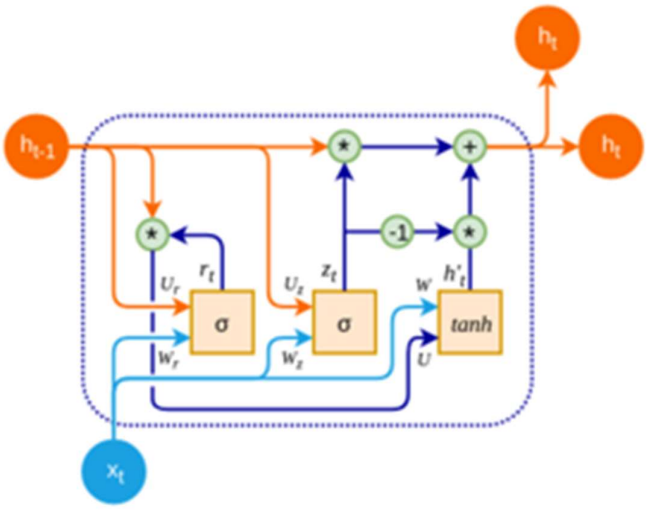



Figure 17. Illustration of a GRU cell. (Nama, 2023)

1. *Update Gate:* $z_t = \sigma(W_z \cdot (h_{t-1}, x_t)$
2. *Reset Gate:* $r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$
3. *Candidate Hidden State:* $\hbar_t = tanh(W \cdot [r_{t-1} \cdot h_{t-1}, x_t])$
4. *Final Hidden State:* $h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \hbar_t$

Where:

- $\sigma$ represents the sigmoid function, which outputs a value between 0 and 1
- $h_{t-1}$ is the hidden state at the previous time step
- $x_t$ is the current input at time step $t$
- $h_t$ is the current hidden state at time step $t$
- $W_z, W_r, W$ are weight matrices that learn the transformations of the input and the previous hidden state

To optimize GRU model performance, the following strategies are helpful:

- **Data Preprocessing**: Normalize the data, handle missing values, and create features like time lags and rolling averages.
- **Model Architecture**: Experiment with different GRU layers and units, use dropout to prevent overfitting, and consider bidirectional GRUs for better performance.
- **Training Techniques**: Implement early stopping, use learning rate scheduling, and experiment with batch sizes for better convergence.

- **Hyperparameter Tuning**: Use grid or random search for hyperparameter optimization and perform cross-validation to ensure generalization.
- **Evaluation and Monitoring**: Use a validation set for monitoring performance and adjust hyperparameters as needed, while evaluating using RMSE, MAE, and $R^2$ (Cho et al., 2014; Kundu & Nasipuri, 2017).

GRUs offer several advantages, including being faster and computationally cheaper than LSTMs, making them ideal for tasks that require long-term dependencies. However, GRUs may not model very long-term dependencies as effectively as LSTMs and can be prone to overfitting, particularly on small datasets. Additionally, they require careful tuning of hyperparameters to achieve optimal performance (Cho et al., 2014; Zhou & Yang, 2015).

Despite these limitations, GRUs remains a powerful tool for sequential data tasks such as time series forecasting and recommendation systems. Their ability to capture temporal dependencies makes them especially suitable for applications like household energy consumption forecasting, where understanding past usage is crucial for predicting future consumption. By following best practices and leveraging GRUs, accurate models can be developed to improve energy efficiency and sustainability (Kundu & Nasipuri, 2017).

## 2.4 TCNs

TCN (temporal convolutional network) is a type of deep learning model that is designed to handle sequential data, such as time series or video frames. TCNs have been applied to a variety of tasks, including anomaly detection, where TCNs have been shown to perform well in detecting anomalies in time-series data.

TCNs have several advantages as well as disadvantages. One advantage of TCNs is their ability to work with variable-length sequences without information leakage due to the causal convolutions. It ensures that the output at any given time depends only on the current and past information with the preserved temporal ordering of the data. Temporal order of the data means that the order or the sequence of the events is followed. TCNs have a simpler architecture, and they are computationally less expensive than RNNs. Additionally, TCNs have several advantages compared to traditional Recurrent Neural Networks (RNNs), including parallel computations, flexibility, and consistent

gradients. The parallelization capabilities of TCNs allow them to train models faster. Another essential advantage is the consistent gradients produced during backpropagation that allow TCNs to learn long-term dependencies effectively. (Auffarth, Machine Learning for Time-Series with Python: Forecast, Predict, and Detect Anomalies with State-of-the-Art Machine Learning Methods, 2022)

On the other hand, TCNs have high memory usage during evaluation due to the large receptive fields created by dilated convolutions. A receptive field is the part of the input sequence that a layer in the network can use to make predictions. In TCNs, the receptive field grows as more layers are added, which helps the model understand patterns over long periods of time. However, this also means that the model needs more memory to process all this information during evaluation. Additionally, TCNs have problems with transfer learning, which can limit the ability to adapt to new tasks (Alla & Adari, Beginning Anomaly Detection Using Python-Based Deep Learning: With Keras and PyTorch, O'Reilly Media, 2019).

TCNs are based on convolutional layers. However, instead of processing images, TCNs are designed to work with sequential data. 1D convolutions are used to capture patterns over time, in other words, they utilize information from previous time steps to make predictions. Compared to RNN, where sequences are processed one step at a time, TCNs consider multiple steps at the same time which makes the training faster (Unit8, 2024). Another essential feature of TCNs is the use of dilated convolutions. Dilated causal convolutions are used to process the input sequence without significantly increasing the depth of the network. For instance, if predicting today's sales requires information from a month ago, dilated convolutions enable the model to efficiently capture recent and distant patterns without increasing computational capacity. All in all, it makes it possible to learn from long sequences while maintaining a manageable model size (Alla & Adari, Beginning Anomaly Detection Using Python-Based Deep Learning: With Keras and PyTorch, O'Reilly Media, 2019).

Another essential feature of TCNs is called the causal convolution. In causal convolutions, the model can only utilize past historical information to predict future values. In causal convolutions, the output at any time step depends on the current and previous information. By combining causal convolutions and dilation, TCNs can utilize long-term dependencies efficiently without requiring an excessively deep network. This approach is particularly useful in time-sensitive applications, such as weather forecasting and financial time-series analysis. Additionally, causal convolutions ensure that predictions remain interpretable by maintaining a clear directional flow of information, critical for understanding model decisions (Bai et al., 2018; Lea et al., 2017).

TCN models use a 1D fully convolutional network (FCN) architecture (Long et al., 2015), where each hidden layer is the same length as the input layer, and zero padding of length (kernel size − 1) is added to keep subsequent layers the same length as previous ones. The TCN architecture uses causal convolutions, convolutions where an output at time t is convolved only with elements from time t and earlier in the previous layer. In other words, the TCN model is the combination of 1D fully convolutional network and causal convolutions (Bai et al., 2018). This ensures that the model maintains the temporal order of the data, which is essential for sequential tasks. Furthermore, TCNs incorporate dilation to efficiently capture long-term dependencies while keeping the computational complexity manageable (Lea et al., 2017). Dilations mean spreading the points that the TCN model looks at in the input sequence. Instead of looking at every single step one after the other (like each frame of a video), the model skips some steps to look at a wider range of the sequence. For example, if the dilation is 2, the model will look at every second step (e.g., step 1, step 3, step 5), and if the dilation is 4, it will look at every fourth step (e.g., step 1, step 5, step 9). In other words, this helps the model look further back in the sequence without extra layers or use a lot more memory.

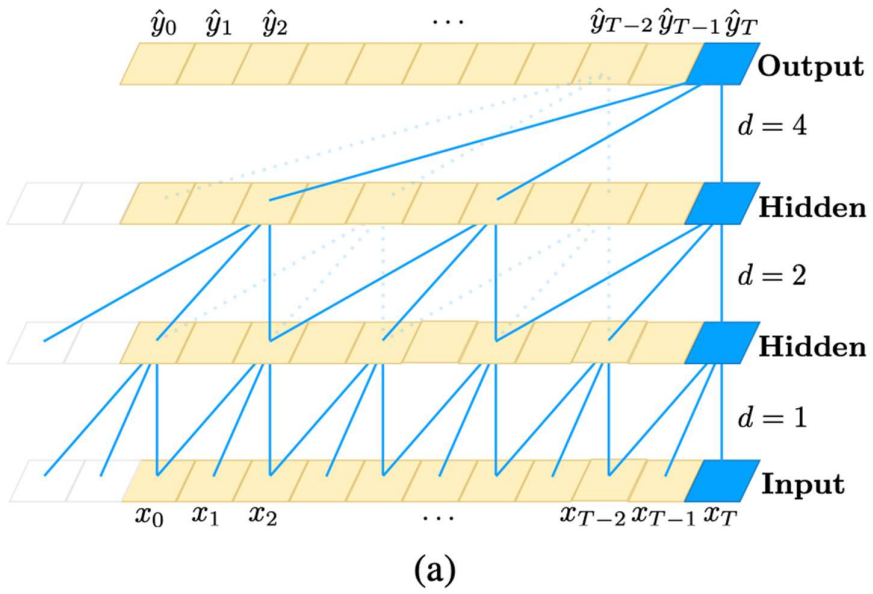



Figure 18. TCN: Dilated Causal Convolution. (Bai et al., 2018)

All things considered, TCN models combine the strengths of 1D fully convolutional architectures, causal convolutions as well as dilations to efficiently model sequential data while keeping the temporal order of information. The use of parallel computations, the possibility to tackle long-term de-

pendencies, and the possibility to stabilize training with residual connections, TCNs can be considered as a powerful alternative to traditional recurrent models for time-series. Nevertheless, the limitations, such as high memory usage during evaluation and challenges in transfer learning are to be considered when deploying TCN models in real-world applications. Despite all these disadvantages, TCNs is a scalable solution for many sequence-modeling problems.

## 2.5 WaveNet

WaveNet, a deep learning model originally developed for audio synthesis, acts as a probabilistic generative model that conditions each data value in a series based on previous values (van den Oord et al., 2016). It models the relationship between past and future values through convolutional layers. WaveNet consists of two convolutional layers: Casual convolution and 1 × 1 convolutions. Casual convolution ensures that any output given at time step $t$ depends only on the current and retroactive values. For example, a time series consisting of $[x_1, x_2, x_3, x_4, x_5]$, a prediction made at $x_3$ will only consider the values in $[x_1, x_2, x_3]$. This provides an advantageous trait when the model is used in time series forecasting due to its ability to maintain temporal structure in a data set. A 1 x 1 convolution performs a linear transformation on the features in a data set at each individual position independently. This means that the model projects input features at each position into a new set of features using weights. Depending on how many features are used in the time series prediction, a 1 x 1 convolution may be necessary to improve the efficiency of a model (Goodfellow et al., 2016).

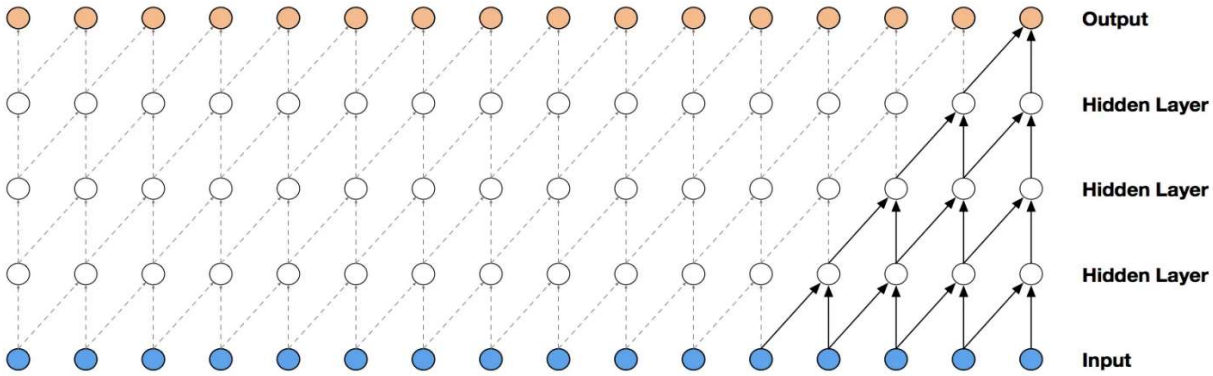



Figure 19. Causal convolutional layer, and the data feed in the NN. (Kumar, 2019)

The figure above is a representation of the convolutional layer and the data feed in the neural network. The output is computed from the retroactive values. A further modification of the convolutional layers is a dilated characteristic where a certain amount of input is dropped based on an arbitrary dilation rate. Dilated convolutional layers help the model capture long-term patterns by

providing exponential growth to the receptive fields, allowing the model to cover a wider range of input values without requiring additional layers.

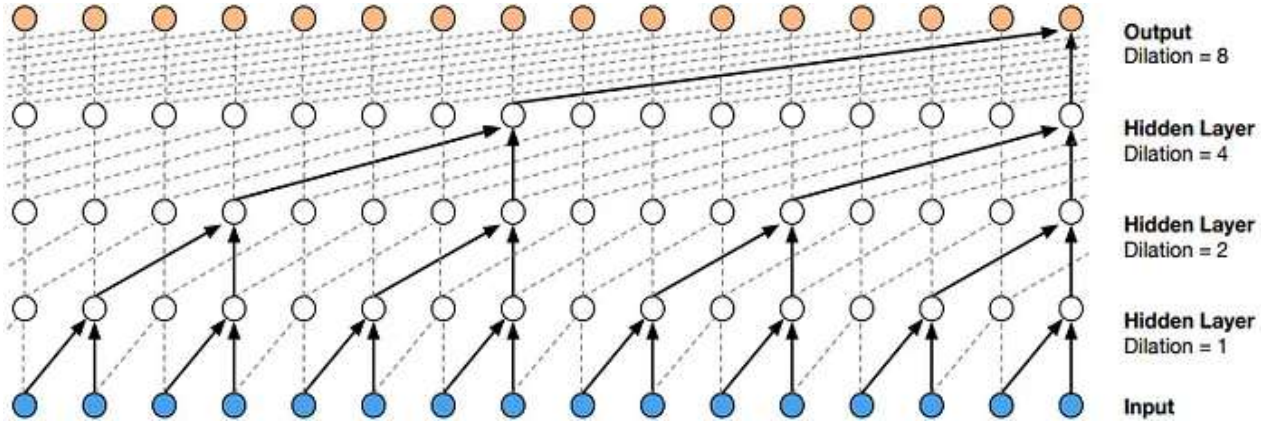



Figure 20. Dilated convolutional layers in the neural network. (Kumar, 2019)

Above is a figure representing the dilated convolutional layers in the neural network. Each subsequent layer exponentially increases the dilation rate. The first layers help the model interpret local patterns and each subsequent layer attempts to teach the model long term patterns. As the dilation rate increases, the dimension of filters is increased by neglecting certain inputs. An additional key feature of the WaveNet architecture is the use of residual connections which structure the data feed between nodes in a format that includes the input and output of the node as a sum. This helps the model preserve data trends from previous layers while also improving the gradient flow and stability since the model learns the residual patterns as opposed to an entire dataset transformation.

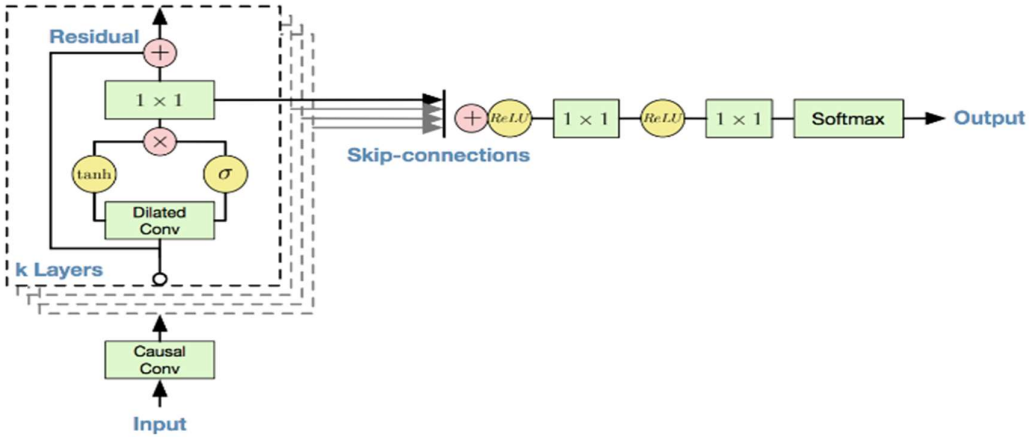



Figure 21. Gated activation in dilated convs: Tanh and Sigmoid Branches. (Kumar, 2019)

In this figure, the dilated convolution splits into two different branches and this would refer to the gated activation element. This is where the convoluted data is fed through a mechanism that determines which part of the transformed data will be passed through. The tanh branch serves as the main transformation of the input where data trends are learnt, and the sigmoid gate branch acts as a filter that dictates how much of the tanh branch is allowed to pass through by combining the two branches using an element wise multiplication.

To break down the mathematical abstraction of the WaveNet neural network architecture, we must first define the input as a time series array.

$$x_t = [x_1, x_2, x_3, x_t]$$

where $x_t$ is value at time t.

The input length is determined based on the kernel size. The sequence is transformed by product and summation of weights and biases. The input sequence must serve as retroactive data so the boundary of the sequence must be determined. Let K be the kernel size (Input size), W the weight of filters, b bias of the layer, and L be the layer index. We can denote the casual convolutional layer output as,

$$z^L(t) = \sum_{i=0}^{k-1} W^L[i] \cdot x(t-i) + b^L$$ (Bai et al., 2018)

Where the variable I denotes the index within the layer. This expression allows us to retrieve a predictive output at time using the boundary set by the summation from i to k-1. The output is only computed with retroactive values provided at a certain time step. This value is transformed by the weight's strength dependent on time step and layer. The product is summed with bias in respect to layer. Next implementation that needs to be made is a dilated characteristic that widens the receptive field by neglecting certain inputs. We will denote the dilation rate as d, and the parameter to be changed is the time step calculation t – i. The following expression can be surmised.

$$z^L(t) = \sum_{i=0}^{k-1} W^L[i] \cdot x(t-d^L \cdot i) + b^L$$ (Bai et al., 2018)

With this modification, certain indexes of x will be skipped based on the dilation rate which increases with each layer. The next infrastructure to describe is the gated activation unit which consists of a tanh branch and a sigmoid branch. The activation functions will encase the output of the

casual convolution layer with the weight and bias transformation. After which both branches will be combined through element-wise multiplication. The output of the gated activation function at given time t can be described as,

$$a^L(t) = \tanh\left(W_f^L * z^L(t) + b_f^L\right) \odot \sigma\left(W_g^L * z^L(t) + b_g^L\right)$$ (Tian & Chan, 2021)

Where subscripts f and g denote the attributes given to the filter and gate. This transforms the output from the convolution layer using the weights and biases with their respective activation functions. Next, we must consider the input of the layer with respect to its output, and this will be the residual connection described. This is a relatively straightforward process where the sum of the input and output of the layer provides the residual connection value expressed as,

$$r^L(t) = a^L(t) + x^L(t)$$ (van den Oord et al., 2016)

The residual output does not explicitly appear in the output function of the network, instead this value is passed on to the next layer. The previous expressions above describe a hierarchical with a certain number of layers L. At the end of each layer a skip value is calculated as the product of the layer's gated output and the skip weight. At the output layer, the summation of skips is utilized. This summation can be described as such.

$$S(t) = \sum_{L=1}^{L} W_s^L \cdot a^L(t)$$ (van den Oord et al., 2016)

With this we can describe the final output layer transforming the skips connection and encasing it with an activation function (SoftMax, linear, etc.). This can be denoted,

$$Y(t) = Activation(W_{out} \cdot S(t) + b_{out})$$ (van den Oord et al., 2016)

## 2.6 Summary

The table presents a comparative analysis of five prominent neural network architectures: **RNN**, **LSTM**, **GRUs**, **TCN**, and **WaveNet**. Recurrent Neural Networks (RNNs) rely on step-by-step sequential processing but struggle with long-term dependencies due to vanishing gradients (Goodfellow et al., 2016). Long Short-Term Memory networks (LSTMs) address this limitation with memory cells and gating mechanisms, making them suitable for long-sequence tasks like speech recognition (Hochreiter & Schmidhuber, 1997). Gated Recurrent Units (GRUs), a simplified version of LSTMs,

retain strong dependency in handling while offering faster training and more memory efficiency (Cho et al., 2014).

On the other hand, Temporal Convolutional Networks (TCNs) and WaveNet leverage fully convolutional architectures, enabling parallel processing of entire sequences. This parallelism significantly accelerates training and enhances their scalability. TCNs use dilated convolutions and residual connections to capture long-term dependencies efficiently, making them ideal for time-series and NLP tasks (Bai et al., 2018). WaveNet extends this capability with dilated causal convolutions and excels in audio and speech synthesis due to its large receptive fields (van den Oord et al., 2016).

Each model balances trade-offs in memory efficiency, parallelism, and long-term dependency handling. RNNs are fundamental for sequential tasks, while LSTMs and GRUs improve dependency learning for complex sequences. Meanwhile, TCNs and WaveNet redefine efficiency and scalability, showcasing the evolution of neural architectures in handling sequential data. This comparison highlights the versatility of these models across diverse applications.

| **Feature** | **RNN** | **LSTM** | **GRUs** | **TCN** | **WaveNet** |
|---|---|---|---|---|---|
| **Architecture** | Recurrent | Recurrent with memory cells | Recurrent with simplified memory cells | Fully convolutional | Fully convolutional |
| **Processing Mode** | Sequential (step-by-step) | Sequential (step-by-step) | Sequential (step-by-step) | Parallel (entire sequence at once) | Parallel (entire sequence at once) |
| **Speed of Training** | Slower due to sequential processing | Slower due to additional complexity | Faster than LSTM, slower than TCN | Faster due to parallelism | Faster due to parallelism |
| **Long-term Dependency** | Struggles due to vanishing gradients | Strong memory cells | Strong but slightly less | Handled using dilated convolutions | Excellent with dilation and |

| | | | | | |
|---|---|---|---|---|---|
| | | | flexible than LSTM | | large receptive fields |
| **Memory Efficient** | Inefficient for long sequences | Less efficient than TCN but optimized for memory | More memory efficient. than LSTM | Efficient, shallow network possible with dilation | Efficient with convolutional operations |
| **Key Features** | Step-by-step recurrence | Memory cells, forget gates, and input gates | Simplified LSTM with fewer gates | Causal and dilated convolutions, residual connections | Dilated causal convolutions, audio/video focus |
| **Parallelism** | Not parallelizable | Not parallelizable | Not parallelizable | Fully parallelizable | Fully parallelizable |
| **Applications** | Sequential data (text or time-series) | Long-sequence tasks (e.g.NLP, speech recognition, time-series) | Short to medium sequence tasks | Time-series, NLP, audio | Audio, speech synthesis, time-series |

Table 1. Comparison of Neural Network Architectures for Sequence Modelling.

# 3 Model Implementation and Results

## 3.1 Common settings for model

The given dataset was split into training, validation sets and test dataset. The dataset was divided as follows:

- Training dataset: 1.1.2016-30.9.2020
- Test dataset: 1.10.2020-30.9.2021
- Eval1: Evaluation 1.10.2021-31.12.2021
- Eval2: Evaluation 1.1.2022-28.2.2022
- Eval3: Evaluation 1.3.2022-31.5.2022

The training dataset provided sufficient data for the model to learn patterns and relationships in the data. The test dataset was reserved to evaluate the model's performance on unseen data after training. To further evaluate the robustness of the model, three additional evaluation periods were defined. Time-lagged features for the target variable (e.g., energy consumption) and time-lagged features of the predictor variable (e.g. air. temperature) were created. Temporal and cyclical patterns, such as the hour of the day, were captured using sine and cosine transformations to allow the model to recognize time-dependent trends. The test dataset was used to measure how well the model generalized to new unseen data with the help of evaluation metrics like Mean Absolute Error (MAE), Root Mean Square Error (RMSE), Mean Square Error (MSE), and R-squared. In addition to testing, the three evaluation periods helped us to access all the model's performance.

The model training was conducted with a maximum of 100 epochs with the early stopping applied. The Adam optimizer was used for training the model and the learning rate was set to 0.001. The Huber loss function was chosen as the objective for training since it is robust to outliers while maintaining sensitivity to smaller prediction errors. It's important to note that the data scaling was not applied to the input features in the final implementation. Initially, we all created models with the scaling applied throughout all the iterations of all models' development, however, it was found that the results showed no significant impact on the models' performance without the scaling of the input variables. Therefore, we decided not to implement the scaling part since it can simply the preprocessing pipeline while still obtaining meaningful results.

**In terms of the main goals, there are 2 main targets in this exercise. These are as follows:**

- Predict the next hour electricity consumption
- Extra: Predict the next day (24h) electricity consumption

## 3.2 RNN

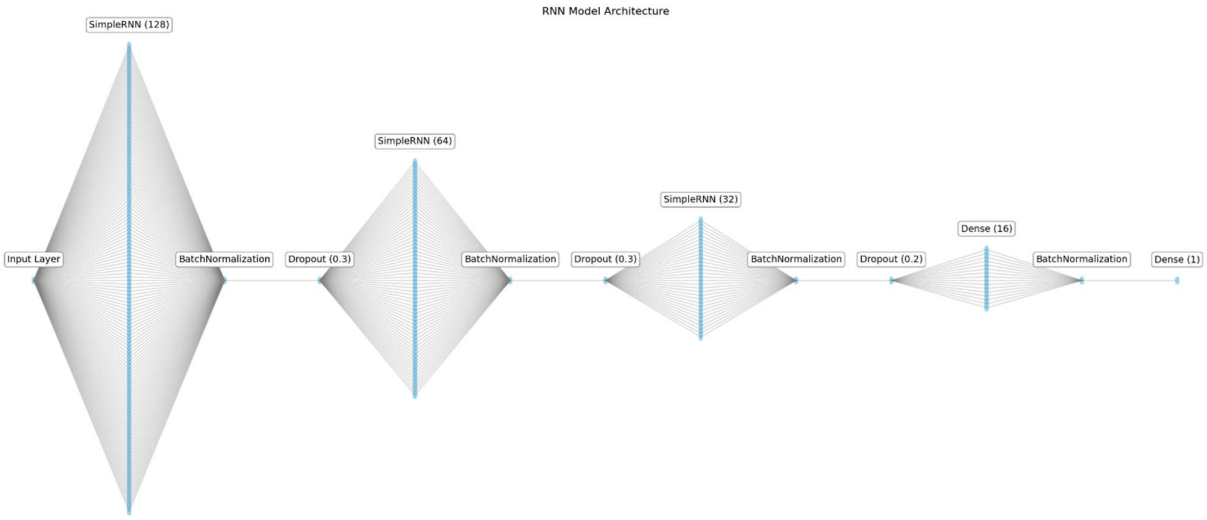



Figure 22. RNN network architecture.

The architecture of RNN network constructed and trained in this group exercise (Appendix 1) is structured to efficiently process multivariate time-series data, integrating specialized layers to capture temporal dependencies and utilizing normalization and regularization methods to ensure stable and effective training. The input layer of the network accepts sequences of shape (sequence_length, num_features), where sequence_length represents the number of time steps and num_features denotes the number of input variables (e.g., energy consumption, temperature). Input features are normalized to the range [0, 1] using MinMaxScaler. This step ensures that all variables contribute equally during training and prevents dominance by features with larger magnitudes.

The RNN comprises three **SimpleRNN** layers, each with tailored configurations to capture both short-term and long-term dependencies in the time-series data:

1. First SimpleRNN Layer (128 units) uses tanh activation function, which maps outputs to a range of [-1, 1], helping to model non-linear relationships. **BatchNormalization** normalizes the outputs to stabilize training by maintaining a consistent distribution across layers. A

dropout rate of 0.3 is applied to randomly deactivate neurons during training, reducing the risk of overfitting.

2. Second SimpleRNN Layer (64 units) also uses tanh activation function, BatchNormalization to stabilize intermediate outputs, and a dropout rate of 0.3 to maintain consistency in regularization.
3. Third SimpleRNN Layer (32 units) utilizes the same tanh activation, and BatchNormalization for stability in the network, but reduced dropout rate of 0.2 to retain more information from the compressed representations.

The dense layers:

1. The first dense layer (16 neurons) uses ReLu (Rectified Linear Unit) activation function providing non-linearity and allowing the model to learn and represent complex relationships in the data. This is crucial for capturing subtle patterns in multivariate time-series data. This is essential for detecting nuanced patterns in multivariate time-series data. Batch Normalization is applied to stabilize and accelerate training by normalizing the layer's inputs. This ensures consistent scaling of features, reducing sensitivity to initialization and improving convergence.

2. The second and the final dense layer (1 neuron) uses linear activation for regression tasks. It maps the learned patterns to a continuous scalar output, such as predicting energy consumption for the next hour.

The RNN model constructed for this exercise follows a training pipeline to address the complexities of multivariate time-series forecasting. The RNN model uses the Adam optimizer, which is dependable and efficient approach for fine-tuning the model during training. The learning rate of 0.001 is used to ensure precise updates. The training process minimizes errors using the Mean Squared Error (MSE) loss function, which focuses on reducing large differences between the predicted and actual values. To provide additional clarity, the Mean Absolute Error (MAE) metric is used, offering a simple and interpretable measure of the model's average prediction error. Techniques like Batch Normalization and dropout layers are applied throughout the network to stabilize training, prevent overfitting, and improve the overall effectiveness of the model.

The RNN model processes multiple input features, allowing it to manage complex datasets effectively. For this application, the target variable is Consumption, while Air Temperature (degC) and its 24-hour moving average are selected as predictors. These features capture both immediate and extended temporal patterns, as well as the impact of weather conditions on energy usage. The sequence length, representing the number of time steps in each input, determines the span of dependencies the model can learn.

To ensure fair contribution from all input variables, the data is normalized to the range [0, 1] using the MinMaxScaler. This step prevents features with larger numerical ranges from dominating the training process. The combination of careful feature selection, preprocessing, and sequence modeling equips the RNN to uncover meaningful patterns in the data, enabling accurate forecasts and valuable insights into temporal dynamics.

The architecture of an updated RNN (Appendix 1), like the previous (original) RNN network, begins with an Input Layer that processes sequences of shape (sequence_length, num_features). This flexible design accommodates multivariate time-series data, where sequence length represents the number of time steps, and num_features denotes the dataset's attributes. Preprocessing ensures all features are scaled to [0, 1] using MinMaxScaler, preventing dominance by variables with larger magnitudes. This setup ensures the model captures temporal dependencies while enabling equal feature contributions during training. The input data is thus primed for advanced sequence modelling.

The core comprises three SimpleRNN layers:

1. The first layer (64 units) captures broad dependencies using tanh activation, L2 regularization (0.007), BatchNormalization, and Dropout (0.3) for stabilization and regularization.
2. The second layer (32 units) builds on this, increasing Dropout to 0.4 for greater robustness.
3. The final RNN layer (16 units) distills patterns into compact representations, followed by BatchNormalization and Dropout (0.4) to ensure stability. These layers progressively refine temporal features.

The Dense layers convert learned sequential representations into actionable predictions:

1. The first Dense layer (16 neurons) employs ReLU activation for non-linearity, supported by L2 regularization to prevent overfitting, with BatchNormalization and Dropout (0.3) further stabilizing learning.
2. The final Dense layer, with a single neuron and linear activation, maps temporal patterns to scalar predictions, completing the sequence-to-value transformation.

The model employs the Adam optimizer, a highly popular and effective method for performing gradient descent. with a learning rate of 0.001 for fine-tuned updates. The Mean Squared Error (MSE) loss function is optimized to minimize the prediction error. Metrics include Mean Absolute Error (MAE), which provides interpretable insights during training. Optimization is further refined using a OneCycleLR scheduler, dynamically adjusting learning rates throughout training to improve convergence. EarlyStopping halts training when validation performance stagnates, preventing overfitting. Together, these strategies create a robust and efficient training pipeline for time-series forecasting.

The model is designed to process multiple features, enabling it to handle complex datasets. In this case, the Consumption variable serves as the target to predict, while Air Temperature (degC) and its 24-hour moving average act as predictors. By including these features, the model accounts for temporal trends and weather-related influences on energy usage. The sequence length determines the time window over which dependencies are modelled, while normalization ensures features contribute equally during training. Scaling the data to the range [0, 1] using MinMaxScaler prevents dominance by variables with larger magnitudes. This feature-engineering and preprocessing pipeline ensures the RNN can effectively learn patterns, enabling accurate predictions and insights into underlying temporal relationships.

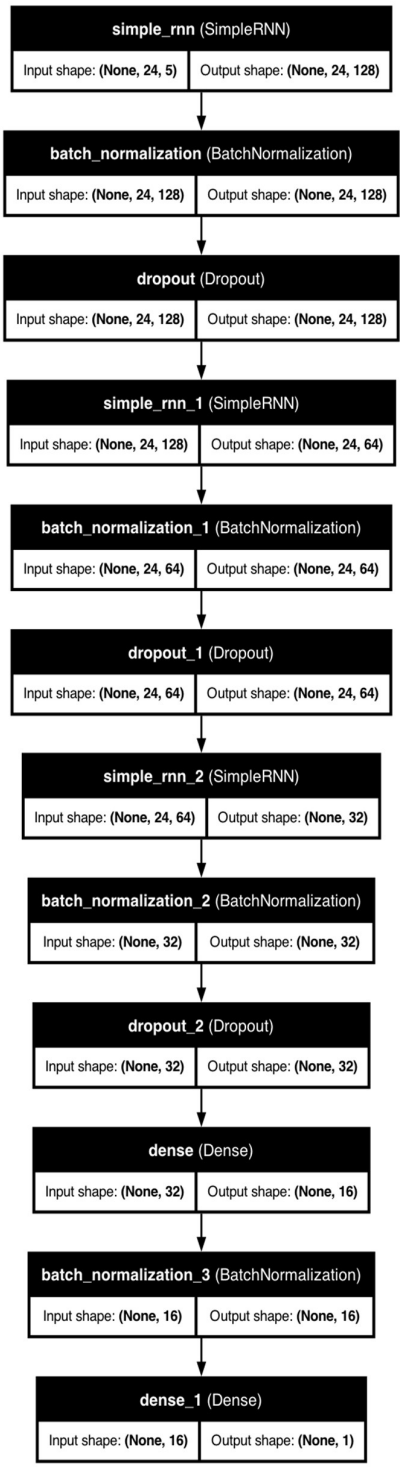



Figure 23. RNN model structure.

The RNN model handles time-series data with an input layer processing 24-time steps (e.g., 24 hours) and 5 features like energy consumption and temperature, enabling the analysis of sequential relationships and variable dependencies.

The architecture (original version) comprises three SimpleRNN layers with 128, 64, and 32 units, capturing temporal dynamics at varying levels. This design enables the model to extract patterns from broad trends to fine details, enhancing its ability to recognize both short-term fluctuations and long-term dependencies.

To ensure stability and accelerate convergence, 4 batch normalization layers standardize intermediate outputs dynamically during training. This approach contrasts with static scaling in feature engineering, allowing the model to learn efficiently.

Dropout layers, placed after each normalization layer, mitigate overfitting by randomly deactivating neurons during training. This prevents reliance on specific features, enhancing the model's robustness to unseen data.

The model's final layers refine data understanding, with a 16-unit dense layer enhancing feature representation. The output layer, a single neuron with a linear activation function, delivers the next hour's energy consumption prediction, ideal for continuous outputs like energy values.

The model employs the Adam optimizer, known for its adaptability to sequential data and dynamic learning rate adjustments, ensuring efficient training convergence. Prediction errors are minimized using Mean Squared Error (MSE), which penalizes significant deviations between actual and predicted consumption values.

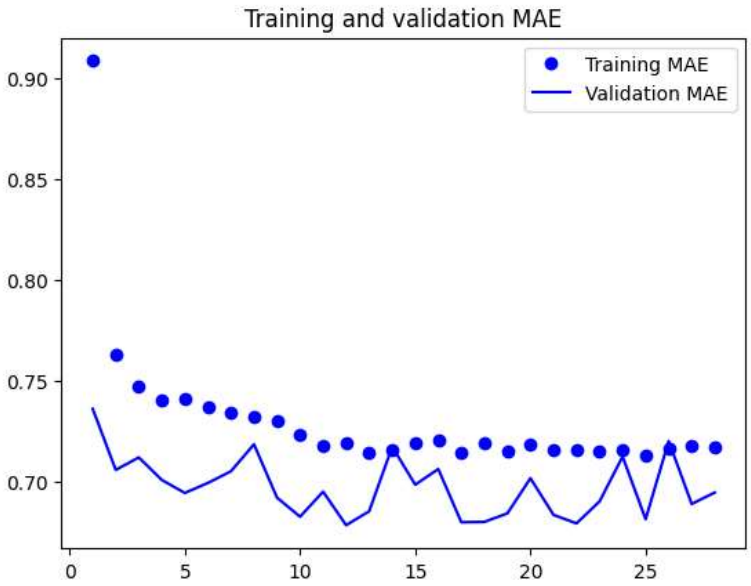



Figure 24. RNN training and validation MAE.

Figure 24 displays four graphs: one for the test set and three for evaluation sets.. The test graph shows that RMSE (0.88) and MAE (0.70) error values are quite low, which suggests that the model performs well on the test data. The predictions align with the actual values. A high $R^2$ value (0.94) indicates that the model explains a significant portion of the variance in the test data. The graph may show predicted values almost overlapping the actual values, with minimal deviation.

Evaluation 1 graph shows that RMSE (0.96) and MAE (0.74) errors are slightly higher compared to the test set but still at a reasonable level. Moderate $R^2$ (0.94) suggests that the model captures some variance, even though the value is only at moderate level, leaving room for improvement. MSE (0.92) indicates moderate squared errors, which suggests some deviations between predictions and actual values. The predictions likely follow the actual values to a reasonable extent, though the deviations are more noticeable than in the test set.

Evaluation 2 graph presents evaluation metrics, showing slightly higher error compared to the test set, being still at acceptable level. Moderate $R^2$ (0.54) suggests that the model captures some variance, significantly underperforming compared to Evaluation 1. MSE (0.92) Indicates moderate squared errors, which suggests some deviations between predictions and actual values. The variance in the data may be partially captured, showing slightly larger prediction gaps.

Evaluation 3 reveals higher errors, RMSE (1.05) and MAE (0.80), than in Evaluation 1, indicating a decline in prediction accuracy. A low $R^2$ suggests that the model struggles to explain the variance in

this evaluation set. A larger squared error: MSE (1.10), indicates greater deviations between predicted and actual values. The model may struggle to identify patterns or fully fit the data in this specific evaluation set, which may be a sign of possible underfitting.

Evaluation 4 graph showcases slightly better results for RMSE (0.91) and MAE (0.72) evaluation metrics as errors are slightly better than Evaluation 2, close to Evaluation 1. $R^2$ (0.17) is like Evaluation 2, in which $R^2$ is low, showing limited ability to explain the variance. MSE (0.83) indicates smaller errors compared to Evaluation 2, suggesting moderate improvement. Predictions may exhibit some alignment with actual values but still have notable deviations. The data may have inherent noise or features that the model has difficulty capturing.

The model performs best on the test set ($R^2$ = 0.94) but struggles with Evaluation 2 and Evaluation 3. This indicates possible overfitting to the training data or differences in data distribution across the evaluation sets. These observations highlight the model's performance variations and point to improvements like reducing overfitting or enhancing generalization.

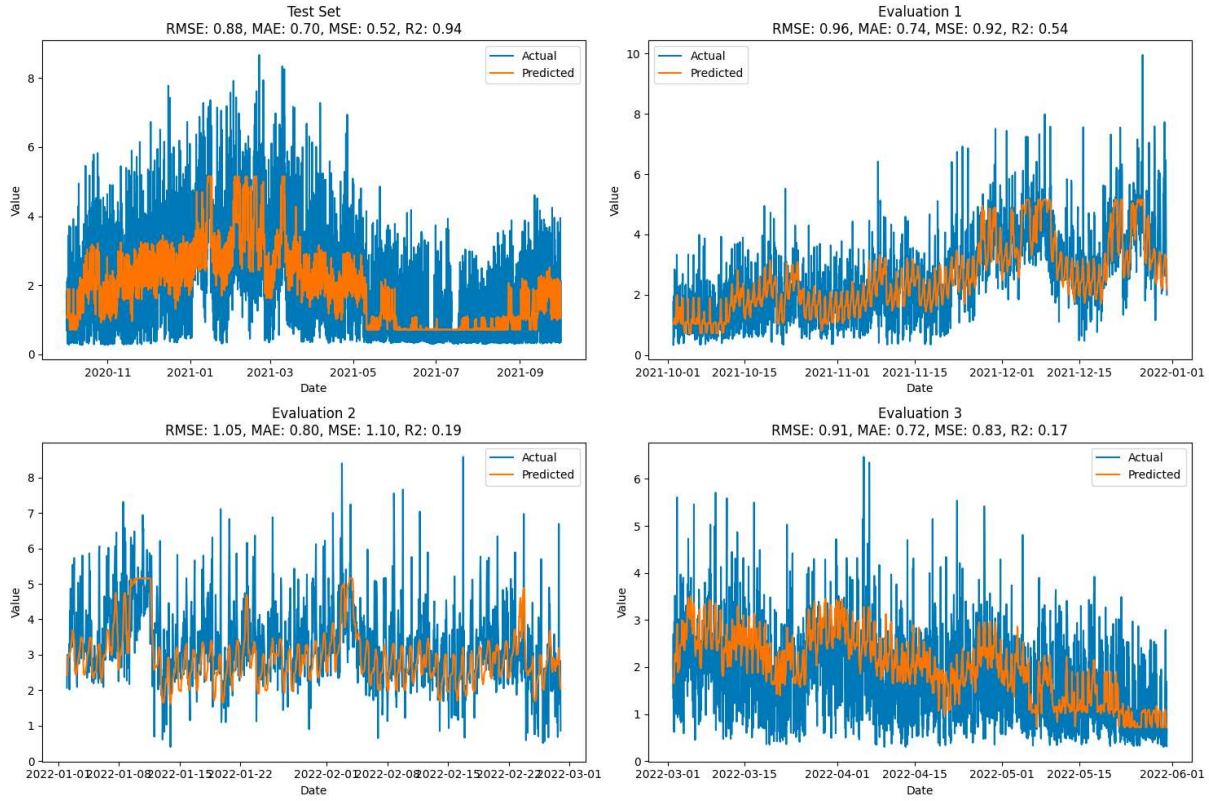



Figure 25. RMSE, MAE, and $R^2$ Analysis for Test and Evaluation Sets.

## 3.3 LSTM

My LSTM development process consisted of two main parts:

- Testing how different features affect the model's accuracy and identifying which features help capture high consumption peaks. It was clear from the first trials that LSTM could predict quite well, but something needed to be developed to better capture peak demands.
- Secondly, the process involved the actual LSTM model building and testing with different hyperparameters.

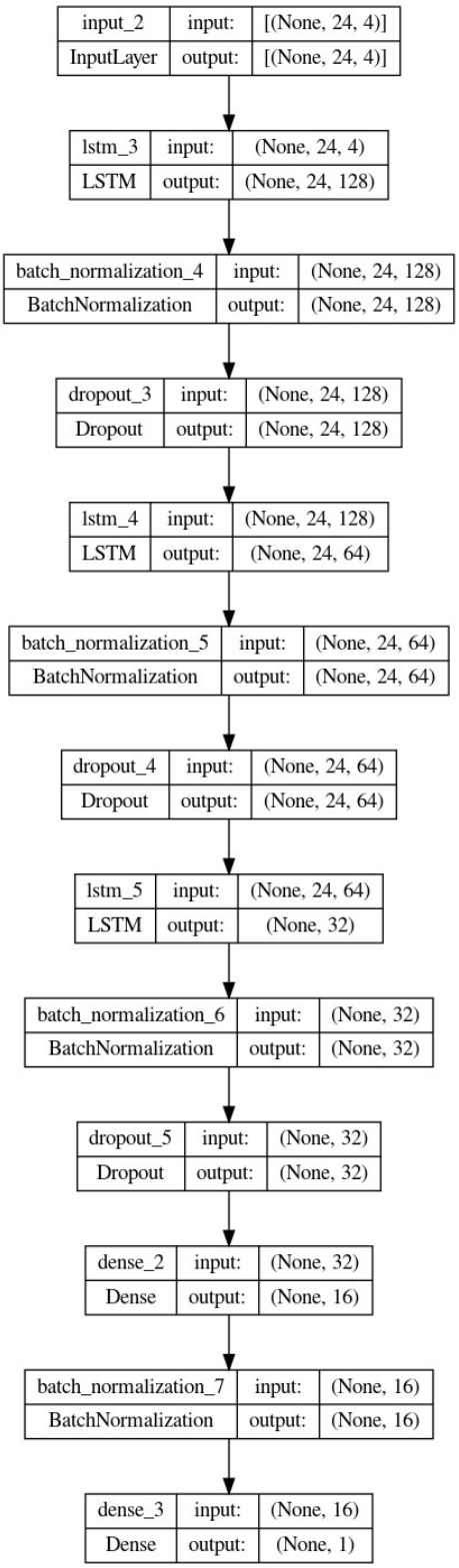
First, I built as simple a model as possible and used only two features: consumption and temperature to predict future values. The predictions were quite accurate, but it was already evident from the visual graphs that the model could not predict peak power consumption accurately. Otherwise, the predicted and actual shapes followed each other quite well.

After the initial trials and errors with the basic model, I decided to start adding more features to the model as described below:

1. I added weekdays as one-hot encoded variables to indicate whether it was a working day or a weekend.
2. In addition, I added a column to predict when the sauna would be ON because it was obvious that the very high peak consumptions were due to the sauna. I added a function to analyze if the power consumption increased rapidly over 3.5 kW and then decreased by 2 kW after one hour. This was especially used between 16:00 and 19:00. In the end, this additional feature did not improve the prediction accuracy significantly and was left out of the final solution to keep it as simple as possible.

Then, I returned to the original idea of keeping the exercise and the model as simple as pos ble. I tested the effects of different hyperparameters and decided to use the EarlyStop function to prevent overfitting. Again, it involved some trial and error to find the right combinations. The final solution was created using our group's common Python script, where everyone's models were implemented to get comparable results. The overview of the LSTM model is introduced on the next page. The actual code is in the appendix

**Model overview**




**Description**

In the input layer of the LSTM-model, 24 represents the length of the input data, for example in this case, 24 hours / repeat data points. The number 4 indicates the number of features (consumption, temperature, sine of hours, and cosine of hours).

There are a total of 3 separate LSTM layers to capture short- and long-term complex patterns from the energy consumption data and other features. These additional layers enhance the model's capability to capture patterns precisely.

The batch normalization layers, which total 4 pcs, normalize the outputs from the LSTM layers to have a mean of zero and a standard deviation of one. This is done in a similar way in the scaling in the feature engineering phase (not part of final implementation in the feature engineering phase). It helps stabilize the learning process and speed up the training.

The dropout layers follow each normalization layer. These are common regularization solutions. During training, they randomly set a fraction of the input units to zero at each update round. The main target is to avoid overfitting to the training data and improve the overall robustness of the model.

The final layer has a single unit because we want to output only one prediction for the next hour's energy consumption. The linear activation function is used for predicting continuous signals like energy consumption.

In this LSTM model, it has been realized that the Adam optimizer works well enough due to its ability to handle complex, high-dimensional data and its robustness.

Figure 26. LSTM model overview

As can be seen from the graph below, it is enough to have around 50-60 epochs. This has been achieved by using the EarlyStopping function to avoid overfitting. During these 50 epochs, the model learns well, and the validation follows the training nicely to achieve perfect training without over-fitting.

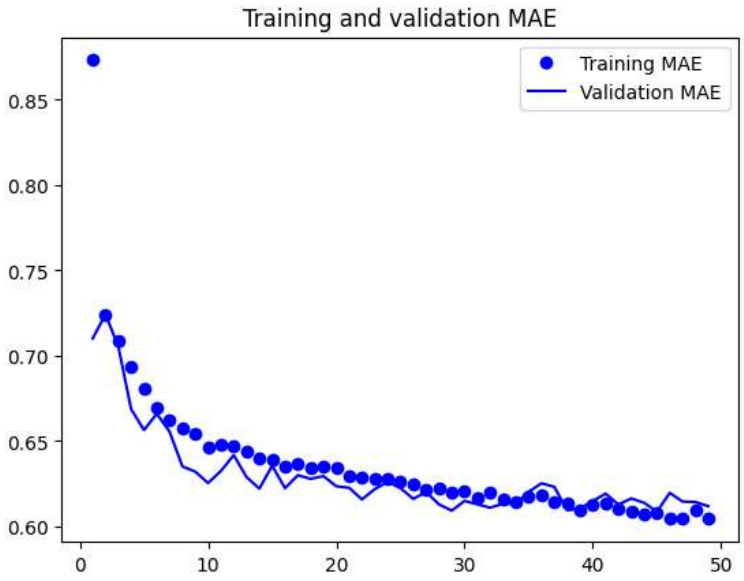



Figure 27. Training and Validation MAE with LSTM model

The graphs below (28) show the LSTM model's performance across the test set and three different evaluation sets. On the test set, the model shows strong predictive capability since the R squared value of 0.85 shows. It means that the model can effectively capture the patterns hidden in the given dataset and explain most of the variability in electricity consumption.

As explained earlier, in my opinion, the LSTM cannot capture all the peak power consumption. For example, in the case of the test dataset, the predicted values are consistently smaller than the actual values. It is not a question of the sauna, but the overall values are lower. It looks like the overall scaling should be multiplied by the factor.

R2 values are best in the case of the Test dataset and Evaluation 1. Evaluations 2 and 3 have higher inaccuracies, and I remember that Mika mentioned during the contact weekend that something strange might have happened during the time of Evaluation 2. This is clearly seen from the KPI values at the top of each graph, with the lowest accuracy in the case of Evaluation 2.

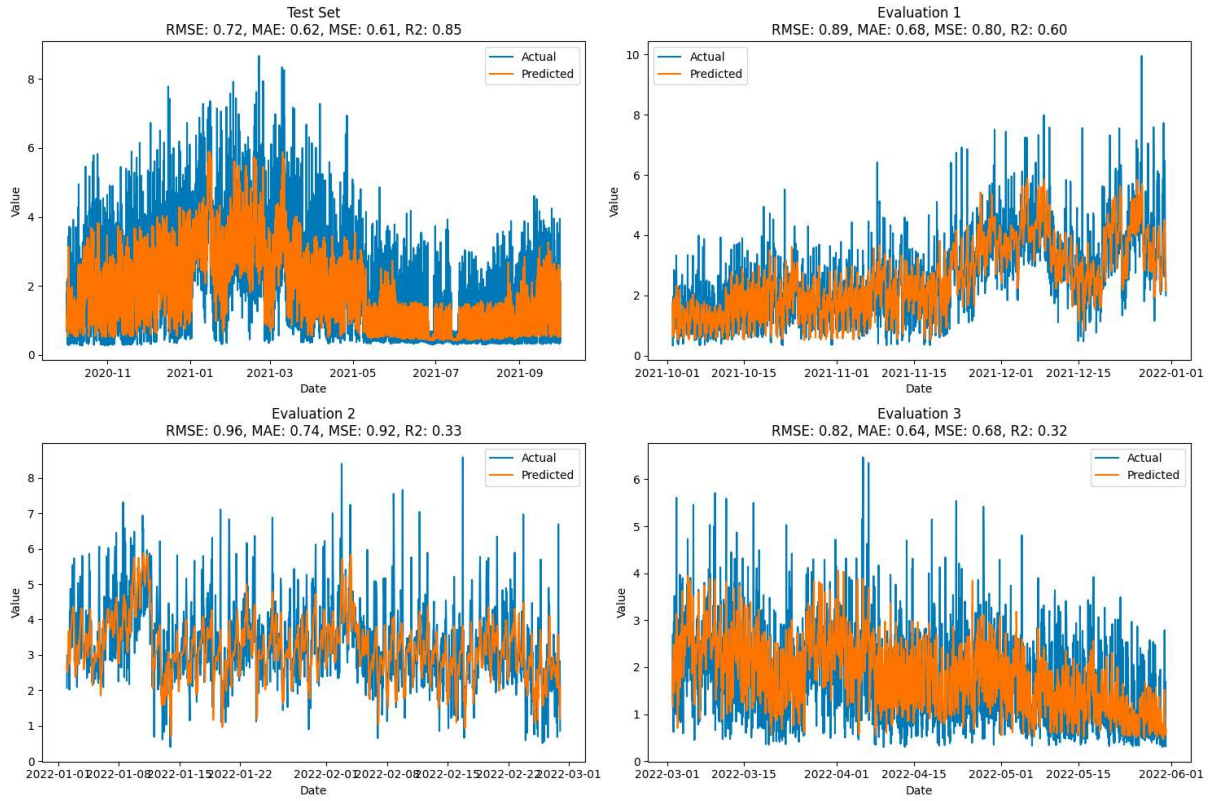



Figure 28. Evaluation of Test Set and Evaluation Sets with LSTM model

## 3.4 GRUs

This section focuses on developing a Gated Recurrent Unit (GRU) model for household energy consumption modelling and prediction. GRU is a type of recurrent neural network (RNN) that is particularly effective for sequential data. In this work, the model will be designed to predict a single output value (consumption) based on a sequence of input features as described in section 3.1.

### 3.4.1 Model Architecture

The GRU model is built using TensorFlow and Keras. The architecture of the GRU neural network is shown in figure 29.

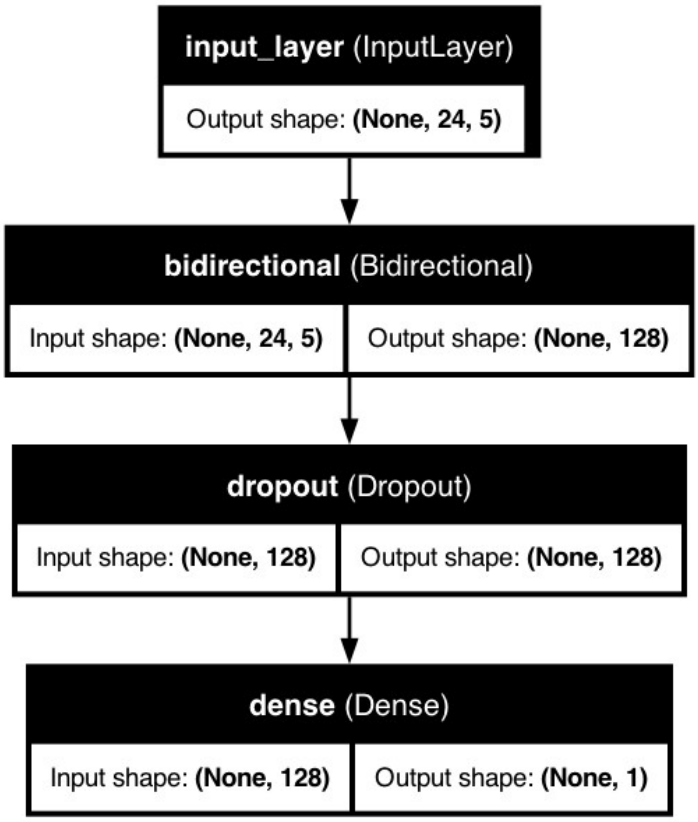



Figure 29. GRU-model structure.

The input layer accepts sequences of data with a specified length (sequence_length) and number of features (num_features). A bidirectional GRU layer with 64 units is used to capture dependencies in both forward and backward directions. This enhances the model's ability to understand the context of the sequence. A dropout layer is added to prevent overfitting by randomly setting a fraction of input units to zero during training. A dense layer with a single unit is used to produce the final output. The model is compiled with the Huber loss function and the Adam optimizer.

### 3.4.2 Benefits of Selected Parameters and Architecture

(a) Bidirectional GRU Layer

By processing the input sequence in both forward and backward directions, the bidirectional GRU layer captures more comprehensive temporal dependencies. This is particularly beneficial for time series data where future values can provide context for understanding past values.

The ability to understand the sequence from both directions often leads to improved prediction accuracy compared to unidirectional models.

(b) Dropout Layer

The dropout layer helps in preventing overfitting by randomly setting a fraction of the input units to zero during training. This forces the model to learn more robust features that generalize better to unseen data. Dropout acts as a regularization technique, ensuring that the model does not become too reliant on specific features or patterns in the training data. A dropout rate value of 0.3 was selected after trial of several other values.

(c) Huber Loss Function:

The Huber loss function is less sensitive to outliers compared to the mean squared error (MSE). This makes the model more robust and reliable, especially in real-world scenarios where data can be noisy. Huber loss combines the best properties of MSE and mean absolute error (MAE), providing a balance between sensitivity to small errors and robustness to large errors.

(d) Adam Optimizer:

The Adam optimizer adjusts the learning rate during training, which helps in achieving faster and more stable convergence. Adam is computationally efficient and requires less memory, making it suitable for training deep learning models. A learning rate of 0.001 was selected. This value resulted in the best accuracy with minimal computational cost.

(e) Training

The model is trained for 100 epochs. The choice of 100 epochs is based on the need to balance training time and model performance. During training, the model's performance is monitored using the mean absolute error (MAE) metric. The training process involves feeding the model with sequences of input data and adjusting the model parameters to minimize the loss function.

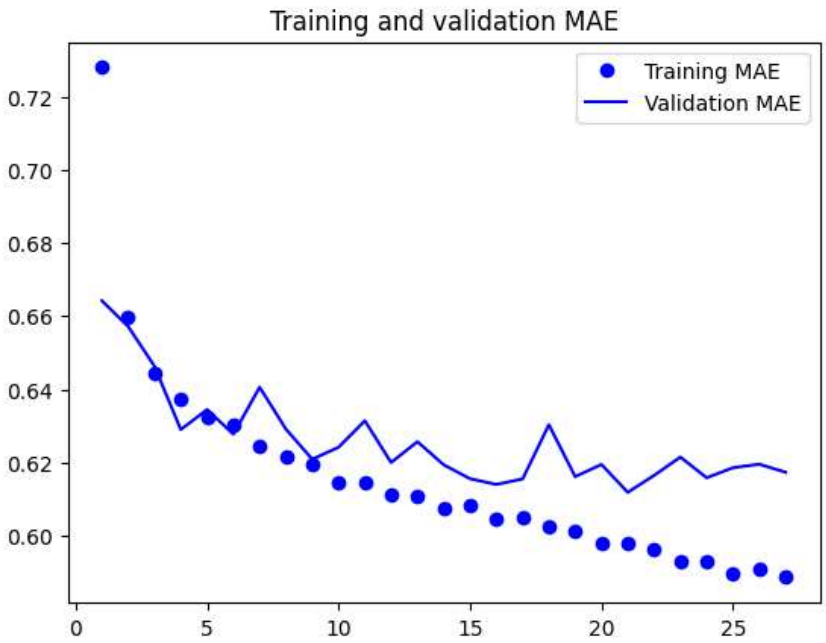



Figure 30. Plot of training and validation MAE loss against epoch for GRU-model.

The figure shows that there is no problem of over-fitting, and the validation MAE does not seem to be reduced anymore after about 25 epochs. Although the training MAE is still dropping.

### 3.4.3 Result and Discussion on GRU Model

The performance of the model is evaluated based on its ability to minimize the Huber loss and MAE. The results indicate that the model effectively learns the patterns in the time series data and provides accurate predictions, but it struggles to capture most of the peak/spikes in the dataset. The use of a bidirectional GRU layer allows the model to capture complex temporal dependencies, while the dropout layer helps in reducing overfitting, leading to better generalization on unseen data.

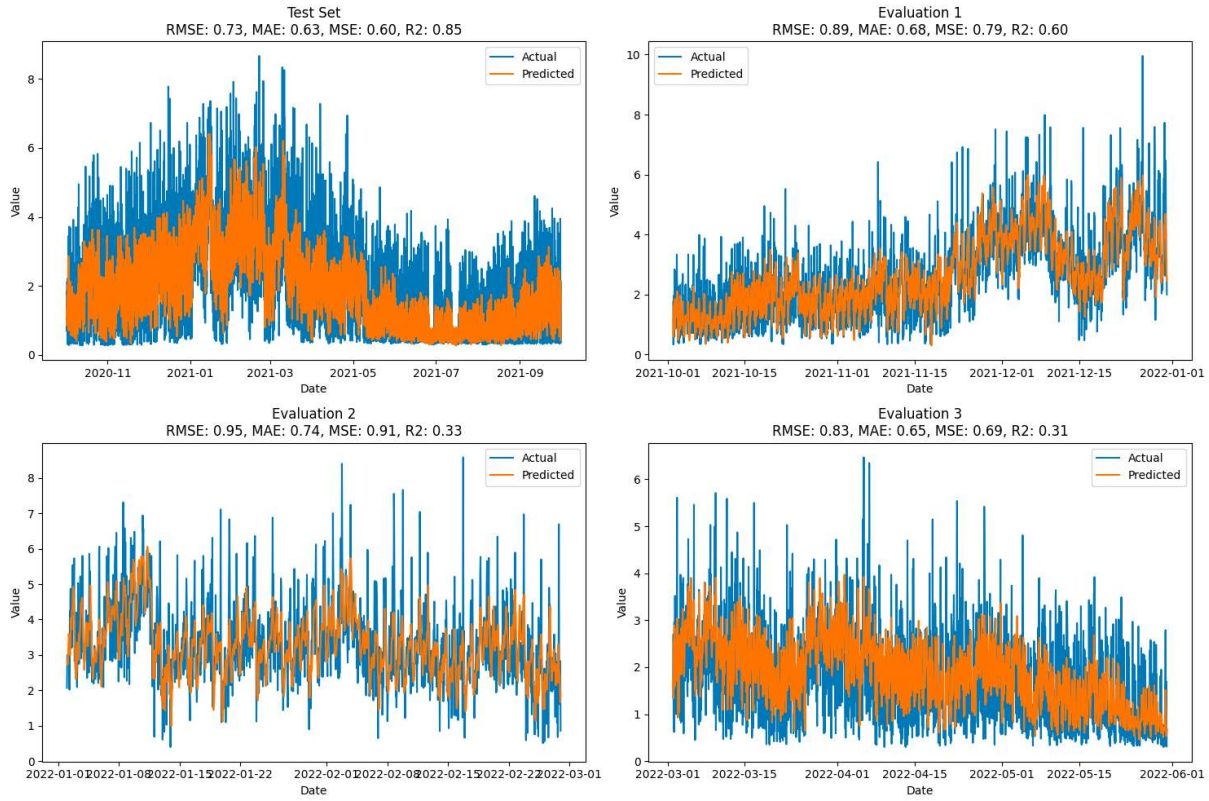



Figure 31. Plot of prediction and actual data for test and evaluations dataset for GRU-model.

To summarize, the GRU model's architecture is designed to handle the challenges of time series prediction, such as capturing long-term dependencies and dealing with noisy data. The bidirectional GRU layer is particularly beneficial as it processes the input sequence in both forward and backward directions, providing a more comprehensive understanding of the temporal context. The dropout layer serves as a regularization technique, ensuring that the model does not become too reliant on any single feature or pattern in the training data. The choice of the Huber loss function is crucial for this project. Unlike the mean squared error (MSE), the Huber loss is less sensitive to outliers, which can be common in time series data at the cost that some of the so-called outliers or spikes will be difficult to capture. Nevertheless, this choice makes the model more robust and reliable in real-world applications. The Adam optimizer, with its adaptive learning rate, ensures efficient and stable convergence during training.

## 3.5 TCNs

There have been multiple iterations of model development. Initially, I created the model using the target variable “Consumption” and one predictor variable “AirTemperature”. “AirTemperature” was selected since it has the high absolute correlation with the target variable along with two other

variables: “MinTemperature” and “MaxTemperature”. The correlation shows that the higher the temperature, the lower the electricity consumption. Including all three variables as predictors is not recommended since it may lead to multicollinearity issues. The second iteration was to add additional features, such as daily (24 hours lag), weekly, monthly and yearly lags of the Consumption variable, as well as “is_weekend” boolean variable and cyclical features of the hour. Based on the correlation matrix, it was found that “is_weekend” and cyclical features of the hour do not have the strong correlation with the “Consumption”, therefore, these variables were excluded. The third iteration was to include only daily (24 hours) lags with the “AirTemperature” variable. The next iteration was to include only daily (24 hours) lags, “AirTemperature” variable and the daily lags of the “AirTemperature” variable as well. Based on the correlation matrix, it was found that the 24 hours lags of the “AirTemperature” variable have a significant correlation with the target variable, therefore, these variables have been included.

As was noticed above in the previous chapter, Temporal Convolutional Network (TCN) models rely on causal and dilated convolutions to capture patterns over various temporal scales, which help them to learn short-term as well as long-term dependencies. Therefore, there are additional parameters that can be fine-tuned while creating a TCN model. As shown in the parameter grid below (Image XX), different hyperparameters of the TCN model were explored using a randomized grid search. These hyperparameters include the number of filters, which define the capacity of the model, and kernel size, which controls the size of the convolutional window used to extract features from the time series. Additionally, dilations were tested with different sets, allowing the model to handle dependencies at multiple temporal scales. Learning rate was also included in the search to balance the speed of convergence, and dropout rate was tested to reduce overfitting by randomly deactivating neurons during training. Finally, batch size was considered to optimize computational efficiency. This parameter grid is used to explore a variety of configurations to determine the best-performing combination for the TCN model.

```
param_distributions = {
    'model__nb_filters': [16, 32, 64],
    'model__kernel_size': [2, 3, 5],
    'model__dilations': [[1, 2, 4], [1, 2, 4, 8], [1, 2, 4, 8, 16]],
    'model__learning_rate': [0.01, 0.001, 0.0001],
    'model__dropout_rate': [0.1, 0.2, 0.3],
    'batch_size': [16, 32, 64]
}
```

Figure 32. Parameter grid usef for the TCN-model.

Once the features were finalized, the RandomizedSearchCV to fine-tune the model's hyperparameters was used. The negative mean squared error (MSE) metric was used in the random grid search to compare the performance of different hyperparameter combinations. The best-found parameters are as follows:

{'model__nb_filters': 16, 'model__learning_rate': 0.001, 'model__kernel_size': 5, 'model__dropout_rate': 0.2, 'model__dilations': [1, 2, 4, 8], 'batch_size': 32}

As can be seen from the image below (XX), the model starts with an input layer that processes sequences with a fixed window size, followed by multiple causal convolutional layers. These causal convolutional layers are needed to ensure that predictions at any time step are influenced only by the current and past inputs, keeping the temporal order. To improve the receptive field, dilated convolutions are applied within these layers, allowing the model to capture short-term as well as long-term dependencies. Each convolutional layer is followed by a dropout layer for regularization and residual connections, that are implemented with the Add layers. The ReLU activations after each residual block introduce non-linearity. Finally, a Flatten layer converts the temporal features into a single dimension, which is passed through a Dense layer to produce the predicted electricity consumption for the next step.

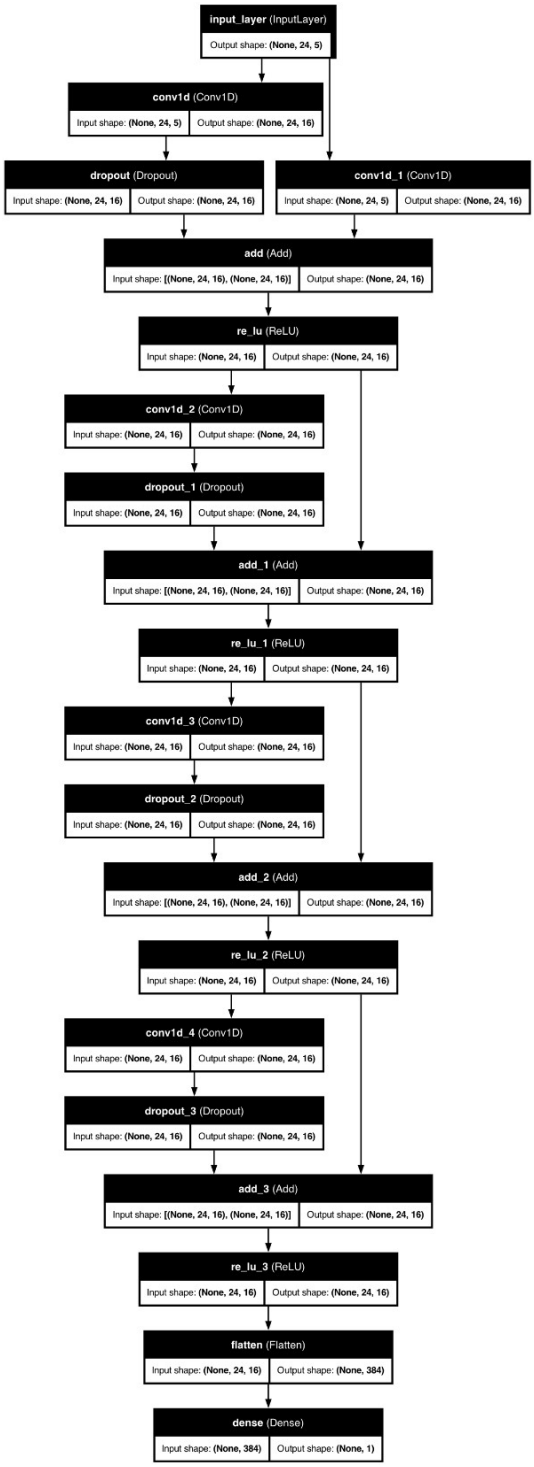



Figure 33. TCN-model structure.

The model was initially trained for a maximum of 100 epochs with early stopping implemented to prevent overfitting. Based on the graph below (XX), one can see that training as well as validation Mean Absolute Error (MAE) steadily decrease and converge to similar values. It means that the TCN model effectively learned the patterns hidden in the dataset.

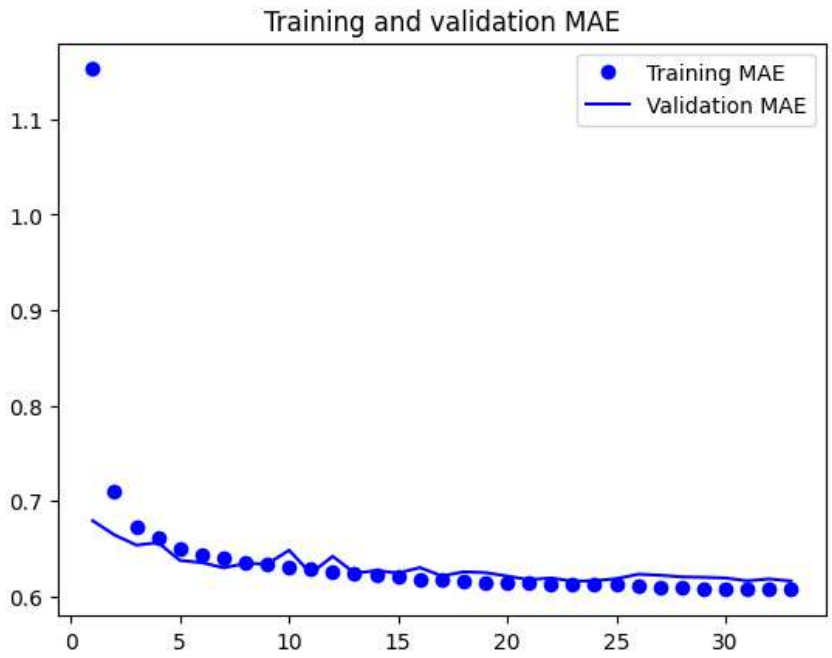



Figure 34. Training and Validation MAE with TCN-model.

The graphs below (Figure 35) show the TCN model's performance across the test set and three different evaluation sets. On the test set, the model shows strong predictive capability since the R squared value of 0.86 shows. It means that the model can effectively capture the patterns hidden in the given dataset and explain most of the variability in electricity consumption. The RMSE of 0.73 and MAE of 0.63 confirm the model's ability to make accurate predictions for the test set. As can be seen from the alignment between the "Actual" and "Predicted" lines in the given graph, the TCN model performs quite well during stable periods when there is a low variability in the dataset. However, during the evaluation periods, the model performance is reduced based on the low value of R squared (0.60, 0.35, and 0.32 for Evaluations 1, 2, and 3, respectively). Based on this information, it can be concluded that the TCN model has difficulties generalizing unseen data, especially in situations where the data differ from patterns observed during the training process. The increased RMSE and MAE values in Evaluations 2 and 3 show larger prediction errors, where we can observe noticeable deviations between the "Actual" and "Predicted" lines. These results indicate possible overfitting to the training set, which can be explained by an unexpected Coronavirus outbreak occurred in the year of 2022. Coronavirus outbreak was unexpected, and it caused the increase of the electricity consumption since people were mostly at home.

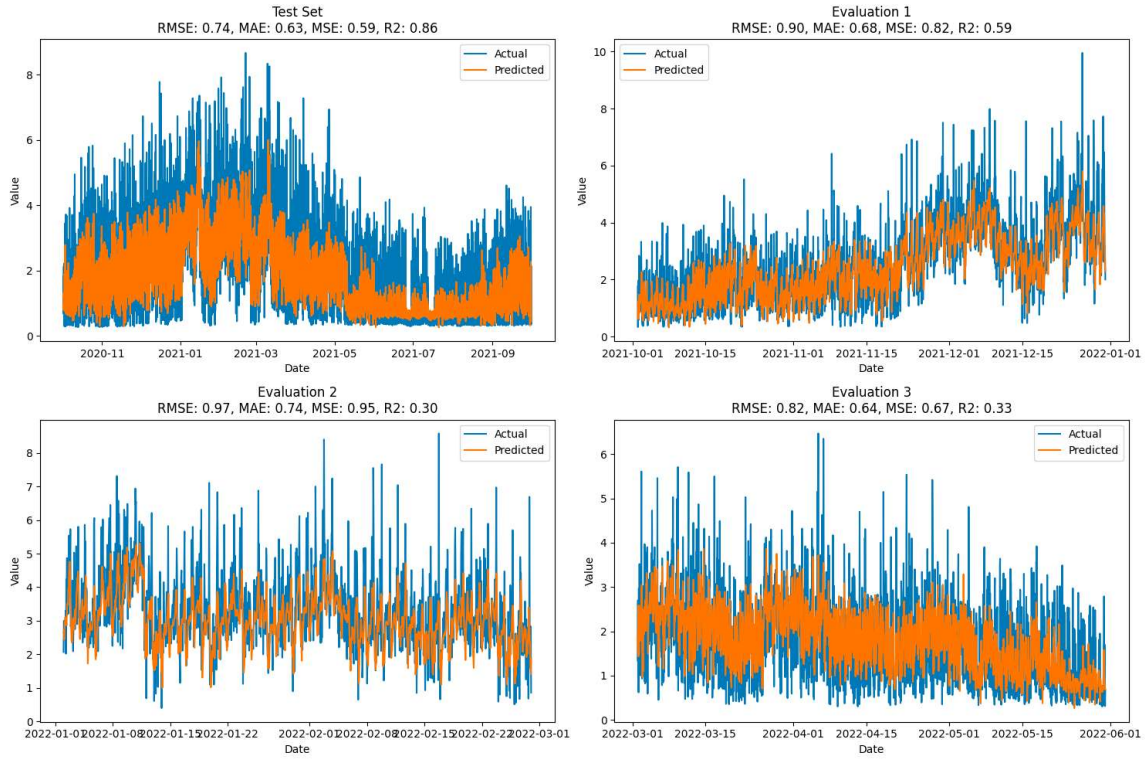



Figure 35. Evaluation of Test Set and Evaluation Sets with TCN-models.

## 3.6 WaveNet

The base model I have created follows the architecture described in section 2.5. First, I will define the dimensions of the filters used in each layer, dilation rate, and input sequence. A list will be used to store the skip connections calculated at the end of each layer. A full overview of the model can be found in appendix 5. The model is not included here due to size constraints.

```
n_filters = 80
filter_width = 3
dilation_rates = [2**i for i in range(9)] * 2

history_seq = Input(shape=(sequence_length, num_features))
x = history_seq
skips = []
```

Figure 36. (WaveNet code snippet).

For this implementation the dilation rate will describe the general architecture of the neural network with regards to its layers. We will perform the transformation at different dilation index which serves as the layer index. Each subsequent layer increases the dilation rate exponentially. We define

the input shape as (sequence_length, num_features) since this will allow for dynamic input shaping with our model allow us to test different domain of features.

```
for dilation_rate in dilation_rates:
    x = Conv1D(32, 1, padding='same', activation='relu')(x)

    f = Conv1D(filters=n_filters,
               kernel_size=filter_width,
               padding='causal',
               dilation_rate=dilation_rate)(x)

    g = Conv1D(filters=n_filters,
               kernel_size=filter_width,
               padding='causal',
               dilation_rate=dilation_rate)(x)

    z = Multiply()([Activation('tanh')(f), Activation('sigmoid')(g)])
    z = Conv1D(32, 1, padding='same', activation='relu')(z)
    x = Add()([x, z])

    skips.append(z)
```

Figure 37. (WaveNet code snippet).

Above is the general transformation structure. The input is first transformed with a 1D convolutional layer. After which a gated activation sequence is implemented with the f and g branches. The output of each branch is then encased in the activation functions tanh and sigmoid respectively which are multiplied together. After which the output is put through another post processing layer the residual connection is then calculated and passed on the next layer. At the end of this sequence the skip connection which is the output of the gated unit is appended to the list defined earlier.

```
out = Activation('relu')(Add()(skips))
out = Conv1D(128, 1, padding='same')(out)
out = Activation('relu')(out)
out = Dropout(0.3)(out)
out = Conv1D(1, 1, padding='same')(out)

pred_next_step = Lambda(lambda x: x[:, -1, :])(out)
```

Figure 38. (WaveNet code snippet).

For the output layer we will be implementing a time distributed dense layer to apply a transformation at each time step independently while also maintaining the temporal structure. First the sum of skip connections is encased in an activation function. After which a 1 x 1 convolution is performed on the output. Another activation function is used after the transformation. To prevent

overfitting, a dropout layer of 0.2 was implemented. The final step is to apply another 1 x 1 convolution that condenses the dimension back to 1 channel that contains the final prediction for each time step as a single scalar value. The next step is then calculated with all the data up until the last time step which will serve as the final output of the model. This same architecture will be used when scaling to a higher list of features along with scaling down to a univariate model. The convolution size and input shape will be adjusted based on the number of features. A univariate model will have its transformation convolution changed to 16 as opposed to 32 while increasing the number of features will increase the dimensions of the convolution to a higher value. Introducing more features also increases the risk of overfitting thus the dropout value will be adjusted as more features are added. In this next section I will be discussing the results I was able to achieve on the three evaluation sets using the model described above. The features used here would be the past electrical consumption, temperature, and the hours of the day as a cyclical feature. The hyperparameters used are 80 filters, 3 filter width, learning rate of approximately 0.001, 10 epochs, 0.3 dropout, and dilation depth of 9. A random search was used to get these hyperparameters thus, potentially improved parameters may exist due to the limited search range.

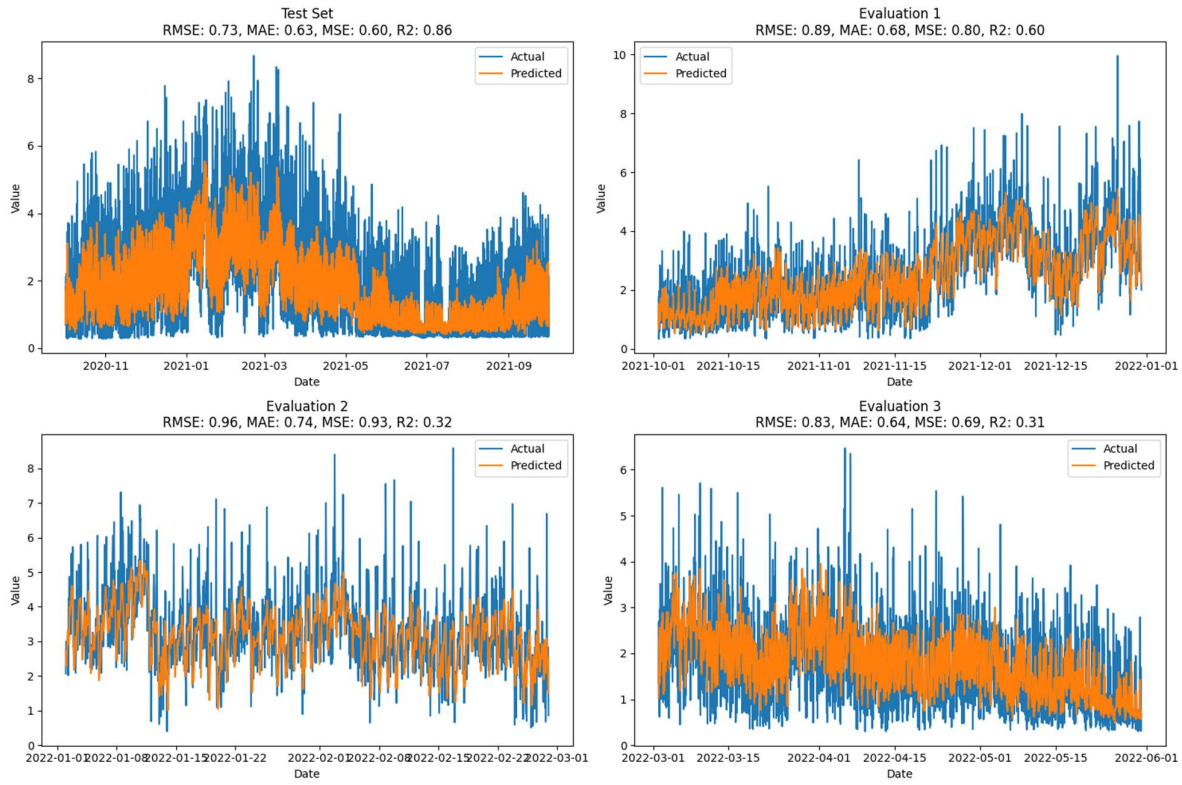



Figure 39. (WaveNet prediction plot).

Looking at the prediction graph for the first evaluation set, we can see that the model is able to capture the general trend relatively well. We can see that it still struggles to capture a lot of the consumption spikes within the data. Overall, the predictions tend to be very steady but fall in the prediction of higher power consumption.

Similarly, the model can capture the general trend of the second evaluation set. We can see though that there is a performance dip relative to the first evaluation set. The logical explanation for this would be that the second evaluation set tends to have more unexpected data values as indicated by the consistent spike throughout the data. Additionally, from the error parameters shown, we can see that the mean squared error seems to be very high. This would align with our earlier explanation that there are a large number of outliers within the data set.

Evaluation set three shows a similar performance as evaluation set two where the model fails to predict many of the large outliers within the data. Generally, the model is still able to follow the trend and capture a pattern. From the error parameters, the high mean squared error would indicate that the data set has a lot of outliers within the data thus causing the lower performance of our model.

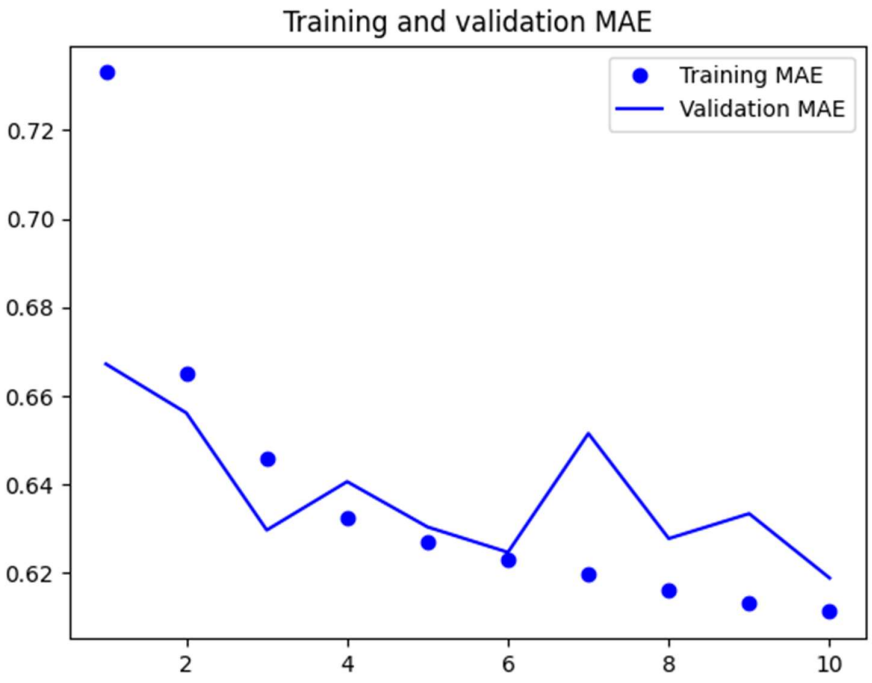



Figure 40. (WaveNet training plot).

When looking at the training pattern, we can see that the training seems to be quite unstable. One potential explanation could be the lower number of epochs the model was trained in, thereby making the discrepancies more apparent. As for the training MAE, we can see that the value seems to lower steadily while approaching a certain asymptote. Initially, an early stopping function was implemented to get the optimal number of epochs. This value came out to 12 epochs, but one thing that can be observed from using 12 epochs is that the model started performing slightly worse at higher epochs. For convenience, I ended up with 10 epoch as the optimal amount since most tests, the best validation loss was found at 10 epoch.

From these results, we can conclude certain characteristics of the model. The model is prone to overfitting at higher epochs due to the model's complexity. Additionally, the model itself is unable to predict outliers due to its single channel characteristics. One potential concept that could be tested is combining two models that have different training emphasis, for example that tries to capture outlier patterns while another captures the general trend. An issue that arises from this system would be determining the predictive distribution of the two combined models, but that will not be discussed here since the focus is of a standalone model.

## 3.7 Summary

Five different models such as RNN, LSTM, GRU, TCN, and WaveNet were created and used with the given dataset to predict electricity consumption. The results of each best possible model can be found in the table below (Table 2). The models are evaluated based on the Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and $R^2$ score. Higher $R^2$ values closer to 1 indicate better performance. As can be seen from the table, the RNN model has a satisfactory performance with a low MAE of 0.70 and MSE of 0.52, RMSE of 0.88, and a relatively high $R^2$ value of 0.94 on a testing set. In terms of the evaluation sets, the RNN model shows a drastic decline in its generalization capabilities, and the $R^2$ values drop to 0.54, 0.19 and 0.17. It shows poor adaptability to new unseen data. The second model, LSTM, has achieved the MAE value of 0.62 MSE value of 0.61, RMSE of 0.72, and an $R^2$ of 0.85 on a testing dataset. In terms of evaluation datasets, LSTM outperformed RNN in evaluation datasets with the $R^2$ scores of 0.60 (Eval1), 0.33 (Eval2), and 0.32 (Eval3). It means that the LSTM model maintains a better balance between accuracy and the model's generalization capability.

| Model | Dataset | MAE<br>Mean Absolute Error<br>(Green best) | MSE<br>Mean Squared Error<br>(Green best) | RMSE<br>Root Mean Squared Error<br>(Green best) | $R^2$ Score<br>(Green/Higher values, closer to 1, indicate better performance) |
|---|---|---|---|---|---|
| RNN | Testing | 0.70 | 0.52 | 0.88 | 0.94 |
| | Eval1 | 0.74 | 0.92 | 0.96 | 0.54 |
| | Eval2 | 0.80 | 1.10 | 1.05 | 0.19 |
| | Eval3 | 0.72 | 0.83 | 0.91 | 0.17 |
| | Average | 0.74 | 0.84 | 0.95 | 0.46 |
| LSTM | Testing | 0.62 | 0.61 | 0.72 | 0.85 |
| | Eval1 | 0.68 | 0.80 | 0.89 | 0.60 |
| | Eval2 | 0.74 | 0.92 | 0.96 | 0.33 |
| | Eval3 | 0.64 | 0.68 | 0.82 | 0.32 |
| | Average | 0.67 | 0.75 | 0.85 | 0.53 |
| GRU | Testing | 0.63 | 0.60 | 0.73 | 0.85 |
| | Eval1 | 0.68 | 0.79 | 0.89 | 0.60 |
| | Eval2 | 0.74 | 0.91 | 0.95 | 0.33 |
| | Eval3 | 0.65 | 0.69 | 0.83 | 0.31 |
| | Average | 0.68 | 0.75 | 0.85 | 0.52 |
| TCN | Testing | 0.63 | 0.59 | 0.74 | 0.86 |
| | Eval1 | 0.68 | 0.82 | 0.90 | 0.59 |
| | Eval2 | 0.74 | 0.95 | 0.97 | 0.30 |
| | Eval3 | 0.64 | 0.67 | 0.82 | 0.33 |
| | Average | 0.67 | 0.76 | 0.86 | 0.52 |
| WaveNet | Testing | 0.63 | 0.60 | 0.73 | 0.86 |
| | Eval1 | 0.68 | 0.80 | 0.89 | 0.60 |
| | Eval2 | 0.74 | 0.93 | 0.96 | 0.32 |
| | Eval3 | 0.64 | 0.69 | 0.83 | 0.31 |
| | Average | 0.67 | 0.76 | 0.85 | 0.52 |

Table 2. Results overview.

The final model ranking has been made by the following way:

1. Calculate the average value of each metric for each model as listed on the above table (Average row).

2. Rank the models for each metric as in the table below (lower is better for MAE, MSE, RMSE and higher is better for R2 Score).

3. Calculate the overall rank by averaging the ranks of all metrics. Right side column on the table below.

| Model | MAE | MSE | RMSE | R2 score | Overall rank |
|---|---|---|---|---|---|
| **RNN** | 5 | 5 | 5 | 1 | **4** |
| **LSTM** | 1 | 1 | 1 | 2 | **1** |
| **GRU** | 2 | 2 | 2 | 3 | **2** |
| **TCN** | 3 | 3 | 3 | 3 | **3** |
| **WaveNet** | 4 | 4 | 4 | 3 | **5** |

Table 3. Overall Ranking ( 1 = best).

The GRU model performed like the LSTM model, and it scored the MAE value of 0.63, MSE of 0.60, RMSE of 0.73 and an $R^2$ value of 0.85 on a testing dataset. Based on the values obtained using the GRU model on all three evaluation datasets, we can see that the GRU model's generalization ability is like the previous model, the LSTM model with the $R^2$ values of 0.60 (Eval1), 0.33 (Eval2), and 0.31 (Eval3). TCN model demonstrated a competitive performance obtaining the following values on a testing dataset: MAE of 0.63, MSE of 0.59, RMSE of 0.74, and an $R^2$ score of 0.86. Although the TCN model performed well on the test set, its $R^2$ scores in evaluation set reduces to 0.59 (Eval1), 0.30 (Eval2), 0.33 (Eval3) like the GRU and the LSTM models. As for WaveNet, the model was able to achieve a $R^2$ score of 0.86 on the testing set while experiencing a drop in $R^2$ value in the subsequent evaluation sets, dropping as low as 0.31 (Eval 3). For the other evaluation sets, a value of 0.60 (Eval 1) and 0.32 (Eval 2) were acquired. The average metric across all sets for WaveNet is 0.67 (MAE), 0.75 (MSE), 0.85(RMSE), and 0.52 ($R^2$). Across all models, these values tend to vary depending on the instance due to the probabilistic nature of these systems.

From these findings, we observe that the LSTM and GRU achieve slightly higher $R^2$ values and lower RMSE values on evaluation sets compared to other models. In comparison with other models, the difference of RMSE and $R^2$ scores is not a drastic one. However, based on the obtained values, we can conclude that the LSTM and GRU models tend to perform better on a new dataset. Additionally, all the models experienced a drastic drop in the $R^2$ scores in the Eval2 and Eval3 sets. This drop can be attributed to the coronavirus pandemic that affected people worldwide and lead to significant changes in the electricity consumption patterns.

Based on the error evaluation created using the Eval1 dataset, we noticed that the errors were the highest during certain days in November and December. Based on the evaluation of errors using the Eval2 dataset, we noticed that the errors were the highest during certain days in January and February. Based on the error evaluation created using the Eval3 dataset, we noticed that the errors were the highest during certain days in March and April. The higher errors observed in November

and December (Eval1) could be due to cold weather since it was observed earlier that the lower the temperature, the higher the electricity consumption. January and February (Eval2) might have high errors because of cold winter conditions, or there are additional features that were not taken into consideration. High errors in March and April (Eval3) could be related to changes in electricity consumption patterns. Most probably, there are additional features that were not considered that could potentially reduce these errors.

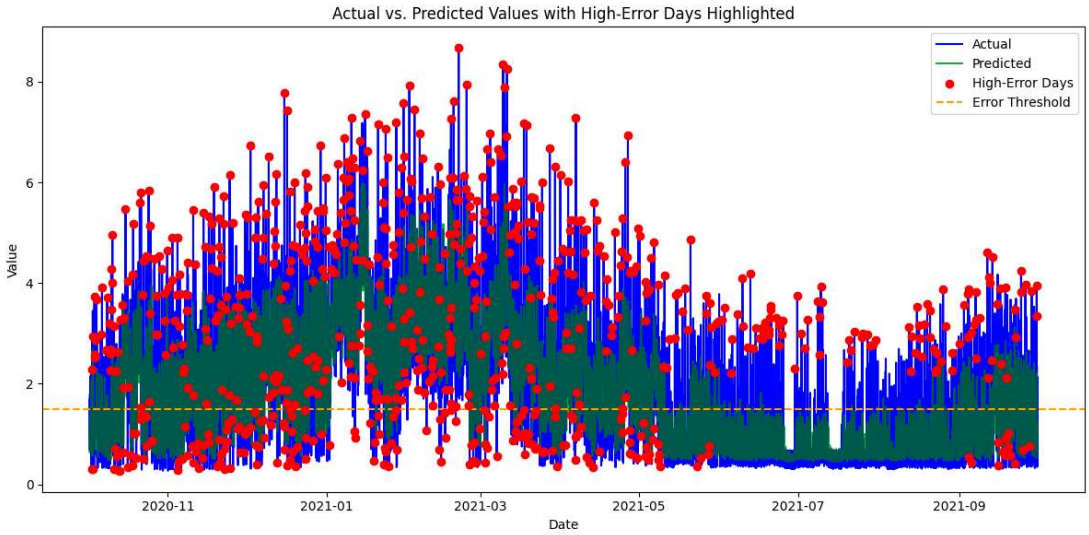



Figure 41. Test Dataset: Actual and Predicted Values with High Error Days using LSTM.

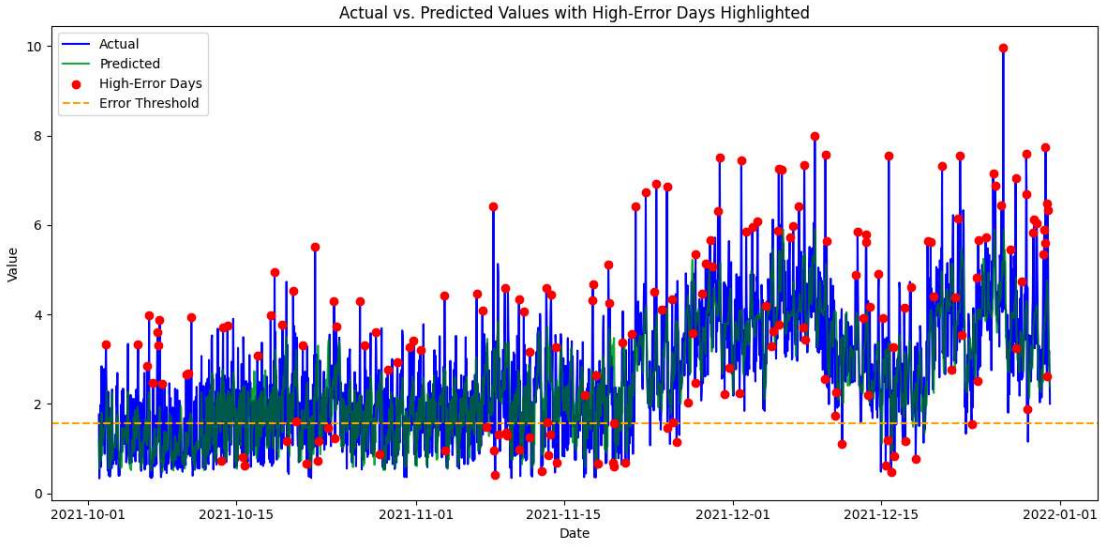



Figure 42. Eval1 Dataset: Actual and Predicted Values with High Error Days using LSTM.

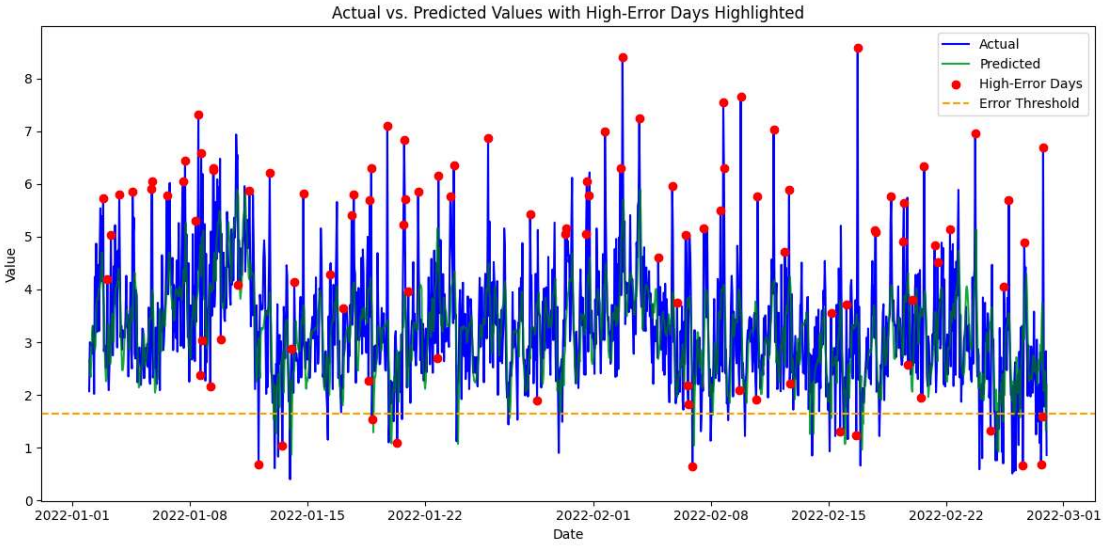



Figure 43. Eval2 Dataset: Actual and Predicted Values with High Error Days using LSTM.

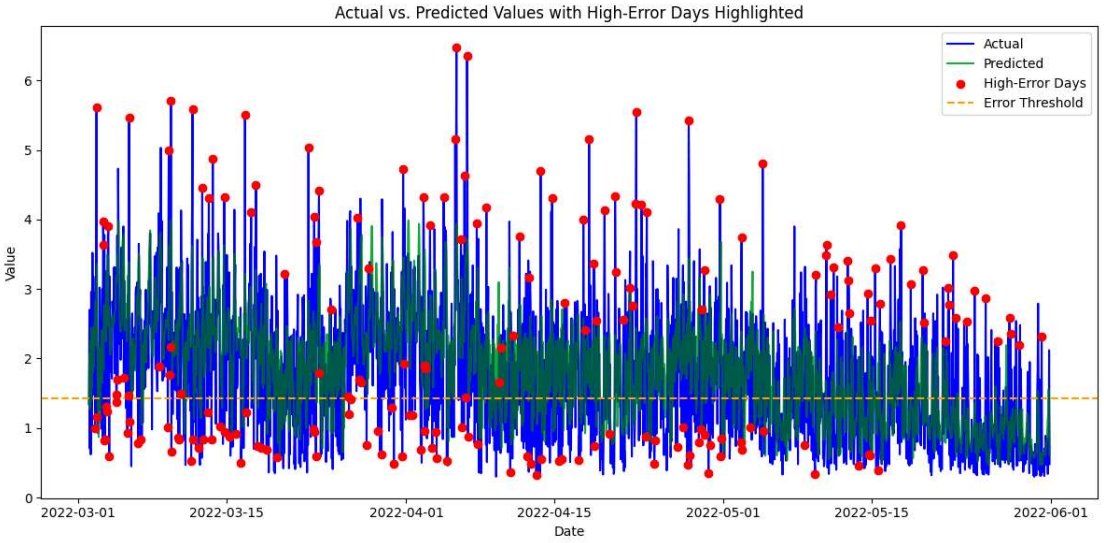



Figure 44. Eval3 Dataset: Actual and Predicted Values with High Error Days using LSTM.

Prediction, Summary and Reflection

The table below summarizes the predictions for the next hour and the next 24 hours from the given dataset i.e. 30.5.2022 at 20.00 pm from all our different models, as this was the main research question in this exercise.

Goal(s):

- Predict the next hour electricity consumption.
- Extra: Predict the next day (24h) electricity consumption.

The predictions from all the models investigated in this study for the next hour of electricity consumption are reported in figure XX. The plot of the next 24 hours consumptions is shown in Figure 37 (a – e).

| Model | Next hour consumption |
|---|---|
| RNN | 0.84 |
| LSTM | 0.71 |
| GRU | 0.74 |
| TCN | 0.74 |
| WaveNet | 0.86 |

Figure 45. Next hourly predictions of the models.

The predictions of the next hour from the models are not too far off from each other except from RNN. This is in line with results from the evaluations dataset where it was observed that RNN and WaveNet's results deviate significantly from the other models on the evaluation dataset. The results of LSTM, GRU, and TCN all produced a range of 0.71 to 0.74.

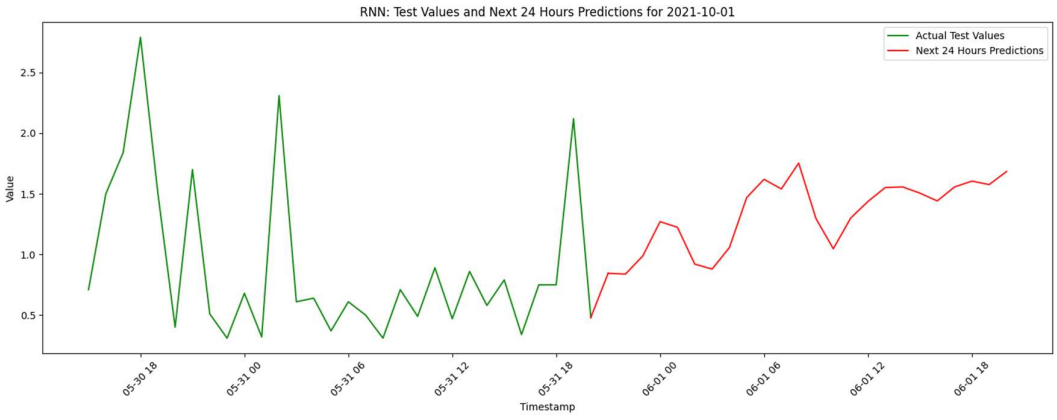



Figure 46. Predictions (a) for the next 24 hours with the RNN model.

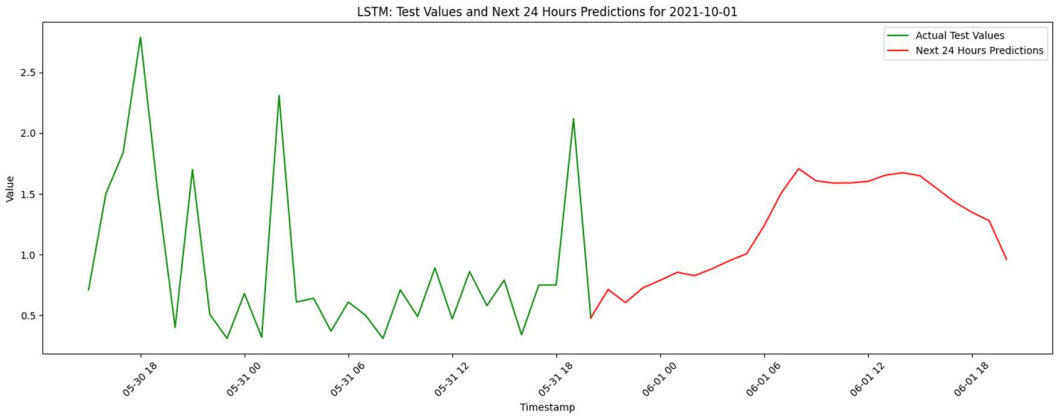



Figure 47. Predictions (b) for the next 24 hours with the LSTM model.

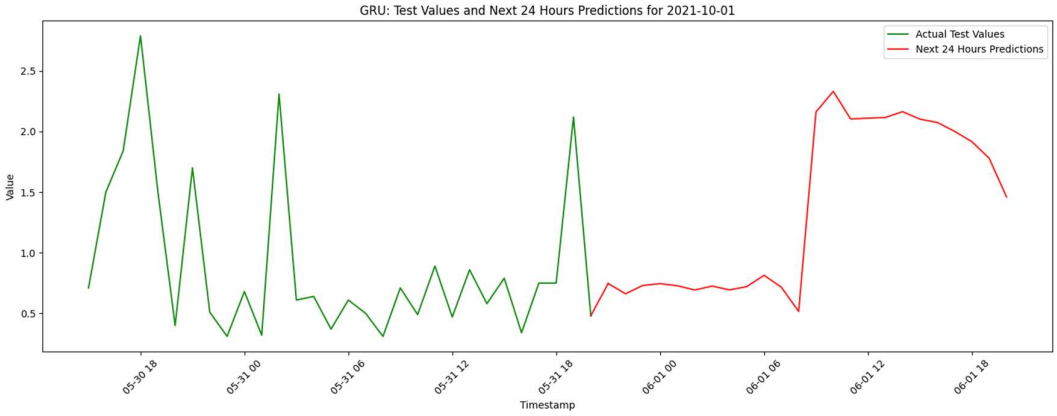



Figure 48. Predictions (c) for the next 24 hours with the GRU models.

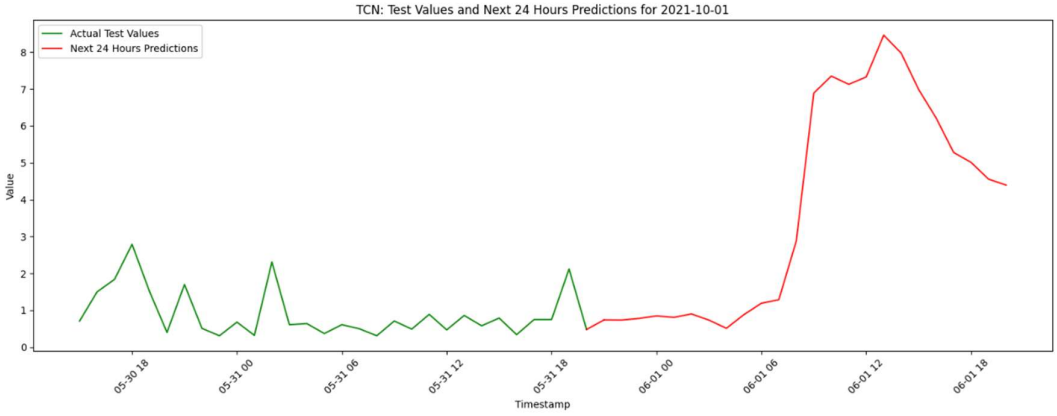



Figure 49. Predictions (d) for the next 24 hours with the TCN model.

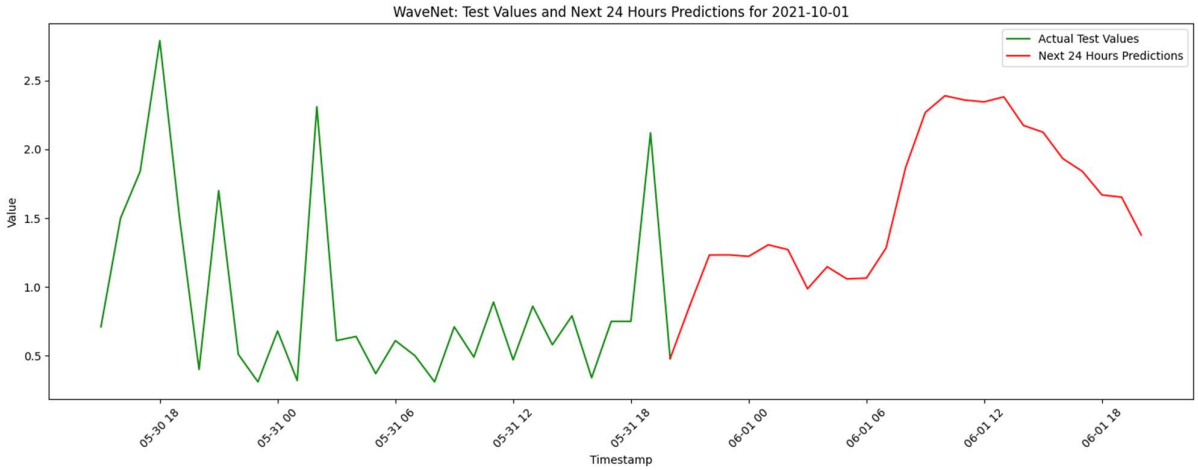



Figure 50. Predictions (e) for the next 24 hours with the WaveNet model.

It is impossible to know which prediction is most accurate since we do not have the actual reading/usage after 20.00 pm on 31.5.2022. Therefore, the prediction from the best performing model on the test, eval1, eval2 & eval3 dataset will be used. The analysis of accuracy scores from the models in section 3.7 shows that LSTM is the best performing model. Hence, the prediction from LSTM is selected as the return of this study is summarized in table 6.

**RESULT (LSTM Model)**

- Predict the next hour (31.5.2022 at 21:00) electricity consumption.

**0.71 KW**

- Extra: Predict the next day (24h) electricity consumption.

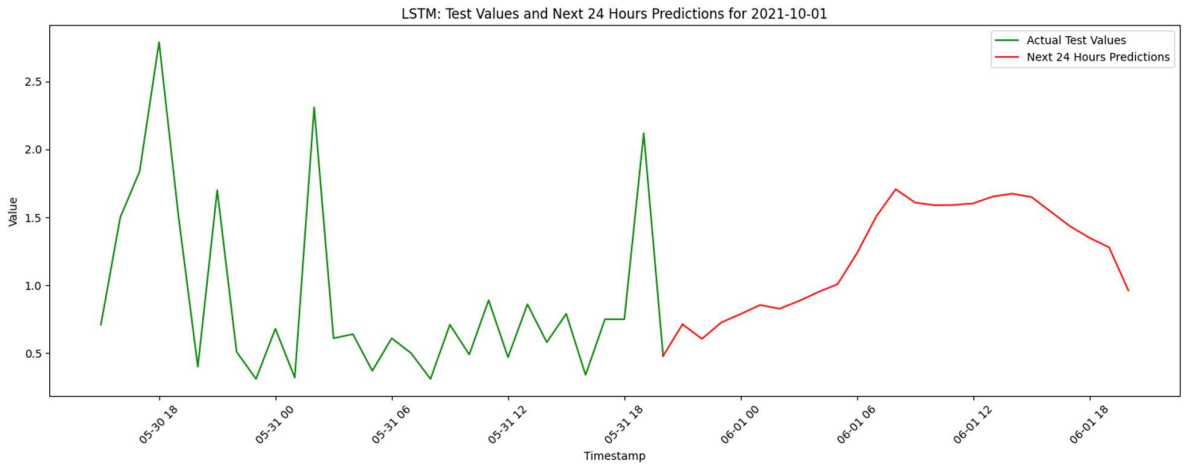



Table 4. Best model predictions.

## References


Alla, S., & Adari, S. K. (2019). *Beginning anomaly detection using Python-based deep learning: With Keras and PyTorch. O'Reilly Media.*

Auffarth, B. (2022). *Machine Learning for Time-Series with Python: Forecast, predict, and detect anomalies with state-of-the-art machine learning methods.* Packt Publishing.

Bai, S., Kolter, J. Z., & Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*. Retrieved from https://arxiv.org/abs/1803.01271

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP 2014)*, 1724–1734.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Kalita, J. (2024). *Architecture of the Recurrent Neural Network*. [Publication details or source to be specified].

Kumar, S. (2019). Understanding WaveNet architecture. *Medium*. Retrieved from https://medium.com/@satyam_kumar/understanding-wavenet-architecture-361cc4c2d623

Kundu, S., & Nasipuri, M. (2017). Gated recurrent units for time series forecasting: A comprehensive survey. *Artificial Intelligence Review, 48*(2), 163–183. https://doi.org/10.1007/s10462-016-9531-1

Lea, C., Vidal, R., Reiter, A., & Hager, G. D. (2017). Temporal convolutional networks for action segmentation and detection. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Retrieved from https://openaccess.thecvf.com/content_cvpr_2017/html/Lea_Temporal_Convolutional_Networks_CVPR_2017_paper.html

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature, 521*(7553), 436–444. https://doi.org/10.1038/nature14539

Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Retrieved from https://arxiv.org/abs/1411.4038

Nama, A. (n.d.). *Understanding gated recurrent unit (GRU) in deep learning*. Medium. Retrieved from https://medium.com/@anishnama20/understanding-gated-recurrent-unit-gru-in-deep-learning-2e54923f3e2

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems, 27*, 3104–3112.

Tian, C., & Chan, W. K. (2021). Spatial-temporal attention WaveNet: A deep learning framework for traffic prediction considering spatial-temporal dependencies. *IET Intelligent Transport Systems*, 15(4), 549–561. https://doi.org/10.1049/itr2.12044

Unit8. (2024). *Temporal convolutional networks and forecasting*. Retrieved from https://unit8.com/resources/temporal-convolutional-networks-and-forecasting

van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., ... & Kavukcuoglu, K. (2016). WaveNet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*. Retrieved from https://arxiv.org/abs/1609.03499

Zhou, Z.-H., & Yang, Y. (2015). Deep learning for time series forecasting: A survey. *International Journal of Machine Learning and Cybernetics, 6*(5), 1697–1715. https://doi.org/10.1007/s13042-015-0391-5

# Appendices – Neural Network codes

## Appendix 1 - RNN

***Original RNN network (graphs etc. in the report):***

```
# RNN Neural Network Code

# Function to define the RNN model
def define_rnn_model(sequence_length, num_features):

    # Create a Sequential model
    model = Sequential([
        Input(shape=(sequence_length, num_features)),  # Input layer
        SimpleRNN(128, return_sequences=True, activation='tanh', kernel_initializer='glorot_uniform'),
        BatchNormalization(),  # Normalize the outputs of the RNN layer
        Dropout(0.3),  # Dropout for regularization
        SimpleRNN(64, return_sequences=True, activation='tanh', kernel_initializer='glorot_uniform'),
        BatchNormalization(),
        Dropout(0.3),
        SimpleRNN(32, activation='tanh', kernel_initializer='glorot_uniform'),
        BatchNormalization(),
        Dropout(0.2),
        Dense(16, activation='relu'),  # Fully connected layer
        BatchNormalization(),
        Dense(1, activation='linear')  # Output layer for regression
    ])
    # Compile the model with Mean Squared Error loss and Adam optimizer
    model.compile(loss=MeanSquaredError(), optimizer=Adam(learning_rate=0.001), metrics=['mae'])
    return model
```

***Updated RNN network:***

```
# Function to define the RNN model
def define_rnn_model(sequence_length, num_features):
    model = Sequential([
        # First RNN layer with L2 regularization and dropout
        Input(shape=(sequence_length, num_features)),
        SimpleRNN(64, return_sequences=True, activation='tanh', kernel_initializer='glorot_uniform', kernel_r
egularizer=l2(0.007)),
        BatchNormalization(),  # Normalize layer inputs
        Dropout(0.3),  # Dropout for preventing overfitting

        # Second RNN layer with L2 regularization and dropout
        SimpleRNN(32, return_sequences=True, activation='tanh', kernel_initializer='glorot_uniform', kernel_r
egularizer=l2(0.007)),
        BatchNormalization(),
        Dropout(0.4),
        # Third RNN layer with L2 regularization
        SimpleRNN(16, activation='tanh', kernel_initializer='glorot_uniform', kernel_regularizer=l2(0.007)),
        BatchNormalization(),
        Dropout(0.4),

        # Dense layer for additional representation power
        Dense(16, activation='relu', kernel_regularizer=l2(0.007)),
        BatchNormalization(),
        Dropout(0.3),

        # Output layer to predict a single value
        Dense(1, activation='linear')
    ])

    # Compile the model with Mean Squared Error loss and Adam optimizer
    model.compile(loss=MeanSquaredError(), optimizer=Adam(learning_rate=0.0015), metrics=['mae'])
    return model

# OneCycleLR-like learning rate scheduler
def one_cycle_lr_schedule(epoch, max_lr=0.0015, epochs=250, steps_per_epoch=567):
    # Calculate total training steps
    total_steps = epochs * steps_per_epoch
    step = epoch * steps_per_epoch

    # Define phase durations
    phase1 = int(0.3 * total_steps)  # Warm-up phase
    phase2 = int(0.7 * total_steps)  # Annealing phase

    # Calculate learning rate based on the current step
    if step <= phase1:
        return max_lr * (step / phase1)  # Warm-up learning rate
    elif step <= phase2:
        return max_lr * (1 - (step - phase1) / (phase2 - phase1))  # Annealing learning rate
    else:
        return max_lr * 0.01  # Decayed learning rate
```

## Appendix 2 - LSTM

```
model = keras.Sequential()
    model.add(layers.Input(shape=(sequence_length, num_features)))
    model.add(layers.LSTM(128, return_sequences=True, activation='tanh', recur-
rent_activation='sigmoid', kernel_initializer='glorot_uniform'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dropout(0.3))
    model.add(layers.LSTM(64, return_sequences=True, activation='tanh', recur-
rent_activation='sigmoid', kernel_initializer='glorot_uniform'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dropout(0.3))
    model.add(layers.LSTM(32, activation='tanh', recurrent_activation='sigmoid',
kernel_initializer='glorot_uniform'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dropout(0.2))
    model.add(layers.Dense(16, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(1, activation='linear'))
```

## Appendix 3 – GRUs

```
inputs = keras.Input(shape=(sequence_length, num_features))
x = layers.Bidirectional(layers.GRU(64))(inputs)
x = layers.Dropout(0.3)(x)
outputs = layers.Dense(1)(x)
model = keras.models.Model(inputs, outputs)
```

## Appendix 4 - TCNs

```
def build_tcn_model():
# These params were found using randomized grid search
# Best Hyperparameters: {'model__nb_filters': 16, 'model__learning_rate': 0.001, 'model__kernel_size': 5, 'model__dropout_rate': 0.2, 'model__dilations': [1, 2, 4, 8], 'batch_size': 32}

nb_filters = 16
kernel_size = 5
dilations = [1, 2, 4, 8]
learning_rate = 0.001
dropout_rate = 0.2
inputs = Input(shape=(sequence_length, num_features))
# Initial stack of dilated causal convolutions
x = inputs
for dilation in dilations:
# Dilated causal convolution
conv = Conv1D(
filters=nb_filters,
kernel_size=kernel_size,
dilation_rate=dilation,
padding='causal',
activation=None
)(x)
conv = Dropout(rate=dropout_rate)(conv)
# Residual connection
if conv.shape[-1] != x.shape[-1]:
x = Conv1D(filters=nb_filters, kernel_size=1, padding="same")(x)
# Add residual connection
x = Add()([x, conv])
x = ReLU()(x)
x = Flatten()(x)
output = Dense(1)(x)
model = keras.models.Model(inputs=inputs, outputs=output)
return model
```

## Appendix 5 - WaveNet

```
n_filters = 80
filter_width = 3
dilation_rates = [2**i for i in range(9)] * 2

history_seq = Input(shape=(sequence_length, num_features))
x = history_seq
skips = []

for dilation_rate in dilation_rates:
    x = Conv1D(32, 1, padding='same', activation='relu')(x)

    f = Conv1D(filters=n_filters,
             kernel_size=filter_width,
             padding='causal',
             dilation_rate=dilation_rate)(x)

    g = Conv1D(filters=n_filters,
             kernel_size=filter_width,
             padding='causal',
             dilation_rate=dilation_rate)(x)

    z = Multiply()([Activation('tanh')(f), Activation('sigmoid')(g)])
    z = Conv1D(32, 1, padding='same', activation='relu')(z)
    x = Add()([x, z])

    skips.append(z)

out = Activation('relu')(Add()(skips))
out = Conv1D(128, 1, padding='same')(out)
out = Activation('relu')(out)
out = Dropout(0.3)(out)
out = Conv1D(1, 1, padding='same')(out)

pred_next_step = Lambda(lambda x: x[:, -1, :])(out)

model = Model(history_seq, pred_next_step)
```