Forecasting Methods – Workshop Manual

1. Introduction

Background

Forecasting involves the prediction of future events or trends based on historical data, current conditions, and various other factors. The prediction of electrical generation and demand through forecasting is essential for network planning and operation of an electrical supply system. Within the context of mini grids, we are interested in performing load forecasting and solar generation forecasting which is considered time-series forecasting. Load forecasting is the process of predicting the amount of electricity that will be consumed by a given population or geographical area over a specific period, typically ranging from a few hours to several years. Accurate load forecasting is a critical component of grid-level energy management, as it helps to ensure that the energy supply matches the demand, while also optimizing the use of resources and reducing costs. Solar or photovoltaic (PV) generation forecasting is the process of predicting the amount of electricity that will be produced by solar power systems, with an aim to optimize the integration of solar power into the overall energy grid. PV generation forecasting models typically rely on a combination of data sources, including historical solar energy production data, weather forecasts, satellite imagery, and other environmental factors. Various methods are used for forecasting tasks in the literature. The most popular methods include machine learning (ML) methods such as artificial neural networks (ANNs), support vector regression (SVR), and statistical methods like ARIMA and regression models [1]–[4].

Outline

This report focuses on the use of neural networks (NNs) and structural time series (STS) models to perform single-step and multi-step forecasting. Many different types of NNs can be used for load forecasting, but this report assesses the performance of deep neural networks (DNNs), convolutional neural networks (CNNs) and recurrent neural networks (RNNs). To build these forecasting models we will use TensorFlow: a free and open-source software library for machine learning and artificial intelligence, with a particular focus on training and inference of deep neural networks. The structure of this report closely follows a couple of TensorFlow tutorials on time series forecasting [5], [6]. The supporting code for this report can be found here [7].

2. Software

Installing Python and Anaconda

Python is a high-level programming language known for its simplicity, ease of use, and versatility. It has a clean and easy-to-read syntax that emphasizes code readability and reduces the cost of program maintenance. Python is used for a wide range of applications, including web development, data analysis, artificial intelligence, scientific computing, and more. Its large standard library and a vast collection of third-party libraries make it a popular choice for both beginner and experienced programmers.

One of the easiest ways to install python is through Anaconda:

1. Go to the Anaconda website at https://www.anaconda.com/products/distribution and download the version of Anaconda that is compatible with your operating system.
2. Once the download is complete, launch the installer.
3. Follow the on-screen instructions to complete the installation process.
4. During the installation process, you will be prompted to choose whether to add Anaconda to your system PATH. It is recommended that you **do not** tick this option.
5. Once the installation is complete, open the Anaconda Navigator application.
6. In the Anaconda Navigator, you can choose which Python environment to use for your projects. By default, Anaconda comes with a base environment that includes Python and many popular libraries. You can also create additional environments for specific projects or tasks. Within an environment,

you can install specific versions of packages without affecting your other environments. Therefore, for this workshop, it is recommended that you create a new environment.

7. To create a new Python environment, click on the "Environments" tab in the Anaconda Navigator, and then click the "Create" button. Give your new environment a name and choose the Python version and any additional packages you would like to include.
8. Once you have created your environment, you can launch a Jupyter Notebook or another Python IDE from the Anaconda Navigator. You can activate your environment in the command prompt or terminal by typing "conda activate <environment name>".

Installing Python Packages

In Python, packages are collections of modules that are used to organize and distribute reusable code. Packages can contain functions, classes, and other objects that can be imported into your Python code to provide additional functionality. The purpose of packages in Python is to promote code reuse and modular programming. Instead of writing all the code for a project from scratch, you can use existing packages and modules to perform common tasks, such as working with files, connecting to databases, or performing complex mathematical calculations. Packages can be installed and updated using package managers such as “pip” or “conda”. Additionally, packages can be versioned, allowing you to use specific versions of a package in your code to ensure compatibility and stability.

1. To install the packages required to run the models, you can use the “Powershell Prompt” application on Anaconda Navigator.
2. Within the Powershell Prompt application, you can activate the project environment in which you want to install the packages using "conda activate <environment name>".
3. From here you have the choice of using “pip” or “conda” to install the packages. Generally, conda leads to more stable installations as it is more thorough in checking and managing the package dependencies. However, in this example, we will opt to use pip to install most packages.
4. To install specific versions of packages using pip, type in “pip install <package name> == <version number>”.

For this workshop, you will need to install the following packages:

- ipykernel
- openpyxl
- matplotlib==3.6.2
- numpy==1.24.1
- pandas==1.5.2
- seaborn==0.12.2
- tensorflow==2.11.0
- tensorflow_probability==0.19.0

3. The Data

The data used in this example contains the historical weather and energy management system (EMS) data for the GionserB mini grid in Kenya. The weather data has been retrieved from the Solcast satellite database. The variables are included in Table 1 below.

Table 1: List of variables

Variable	Unit	Description
Global Horizontal Irradiance (GHI)	W/m^2	The total irradiance received on a horizontal surface. It is the sum of direct and diffuse irradiance components received on a horizontal surface
Direct (Beam) Horizontal Irradiance (EBH)	W/m^2	The direct irradiance (arriving in a straight line from the sun) received on a horizontal surface.
Direct Normal Irradiance (DNI)	W/m^2	The direct irradiance (arriving in a straight line from the sun) received on a surface held perpendicular to the sun.
Diffuse Horizontal Irradiance (DHI)	W/m^2	The diffuse irradiance received on a horizontal surface.
Zenith	Degrees	The angle between a line perpendicular to the earth's surface and the sun (90 deg = sunrise and sunset; 0 deg = sun directly overhead).
Azimuth	Degrees	The angle between a line pointing due north to the sun's current position in the sky. Negative to the East. Positive to the West. 0 at due North.
Cloud Opacity	%	The measurement of how opaque the clouds are to solar radiation in the given location (0 = no cloud, 100 = full attenuation of incoming light).
Air Temp	Celsius	The air temperature (2 meters above ground level).
Dew Point	Celsius	The air dewpoint temperature (2 meters above ground level).
Relative Humidity	%	The air relative humidity (2 meters above ground level).
Sfc Pressure	hPa	The air pressure at ground level.
Wind Speed	m/s	The wind speed (10 meters above ground level).
Wind Direction	Degrees	The wind direction (10 meters above ground level). 0 is a northerly wind.
Precip Water	kg/m^2	The total column precipitable water content.
Snow Depth	cm	The snow depth liquid-water-equivalent.
GTI (Global Tilted Irradiance) Horizontal Single-Axis Tracker	W/m^2	The total irradiance received on a sun-tracking surface.
GTI (Global Tilted Irradiance) Fixed	W/m^2	The total irradiance received on a surface with a fixed tilt. The tilt is set to latitude of the location.
Albedo	-	Average daytime surface reflectivity of visible light, expressed as a value between 0 and 1. 0 represents complete absorption. 1 represents complete reflection.
Ppv1	W	PV Power
Upv1	V	PV Array Voltage
Ua	V	Voltage Phase-a
Ub	V	Voltage Phase-b
Uc	V	Voltage Phase-c
Fac	Hz	Frequency
Preal1	W	Phase-a Active power demand
Preal2	W	Phase-b Active power demand
Preal3	W	Phase-c Active power demand
EMS_total_load	W	Total Active power demand
Pbat	W	Battery Power
SoC	%	Battery State of Charge
Batv	V	Battery Voltage
Batc	A	Battery Current
InvBatV	V	Inverter Voltage

4. Neural Network Models

A neural network (NN) model contains layers of interconnected nodes, each of which performs a mathematical operation on the input data. The output of each layer is then passed to the next layer, where it is processed again. By combining the outputs of all the layers, the model can make predictions about new data. The process of training a NN model involves feeding it a large amount of labelled data and adjusting the weights of the connections between nodes until the model can accurately predict the correct output for each input. Once the model has been trained, it can be used to make predictions on new, unlabelled data.

Data Pre-processing

Missing Data

A selection of key features is plotted over 20 days in Figure 1, in which we can observe large quantities of missing data for the electrical features. To make the dataset usable for the NNs, these missing values must be filled which can be achieved through a method called imputation. We can do basic linear interpolation which draws straight lines across the missing data gap, shown in Figure 2. However, we can use a more advanced method called multivariate imputation by chained equation (MICE) [8] that produces much smoother interpolations, shown in Figure 3.

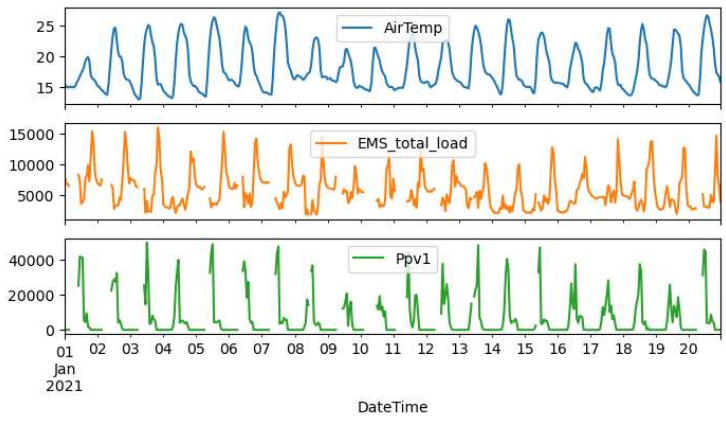

Figure 1: A plot of AirTemp, EMS_total_load and Ppv1 over the course of 20 days.

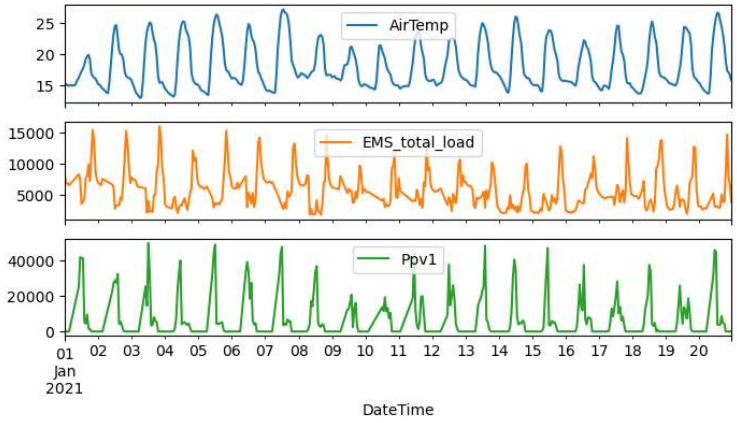

Figure 2: Linear interpolation of data.

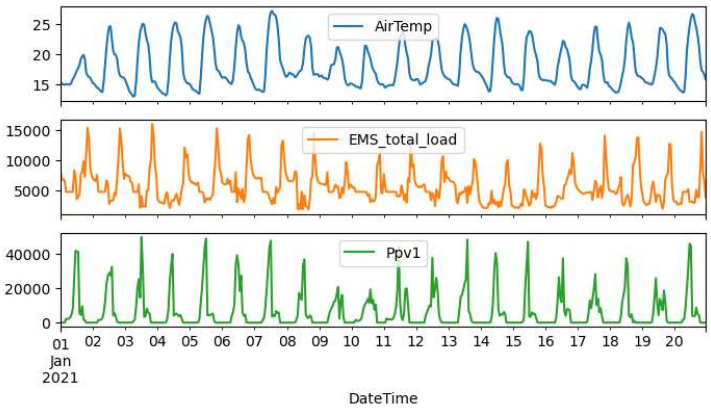

Figure 3: Multivariate imputation by chained equation (MICE).

Test-validation-train Data Split

The use of separate training, validation, and testing datasets is important to ensure that NN models can generalise to new data and perform well in a real-world application. The training dataset is used to learn the relationships between the input data and the corresponding output. During the training process, the model adjusts its parameters to minimize the error between its predicted outputs and the actual outputs in the training dataset. The validation dataset is used to fine-tune the model and optimize its performance with the aim of improving the generalisability of the model. The testing dataset is the set of data that is used to evaluate the final performance of the trained model. In this example, an (80%, 10%, 10%) split is chosen for the training, validation, and test datasets. In addition, unlike in other applications of NNs, the datasets are not randomly shuffled before splitting since the structure of consecutive samples must be maintained.

Data Standardisation

Data is often standardised before being used to train a NN model. Standardisation transforms the input data so that it has a mean of zero and a standard deviation of one. The data is standardised using

$$Z = \frac{x - \mu}{\sigma} \tag{1}$$

where Z is the standardised value (also known as the standard score) and μ and σ are the mean and standard deviation for the training dataset, respectively. Standardisation is important because it ensures that the input features are on the same scale, which can help the neural network to converge faster during training. Additionally, standardisation can also help to improve the accuracy and stability of the NN model by reducing the impact of outliers in the input data.

Data Windowing

Data windowing is the process of dividing a time series into multiple windows, where each window contains a fixed number of time steps. The purpose of data windowing is to transform the time series into a format that can be used as input to a machine learning model. The main features of a data window are the width of the input and label windows, the time offset between them, and the choice of features as inputs, labels, or both. For instance, to make a single prediction 24 hours into the future, given 24 hours of history, a window could be defined as depicted in Figure 4. In the code, the ***WindowGenerator*** class is used to create a set of data windows for a given input width, label width, shift and label features. We can also define a ***split_window*** method that converts the window into a window of inputs and a window of labels. To be able to feed the data into our models, we convert the data into a ***tf.data.Dataset*** using the ***make_dataset*** method.

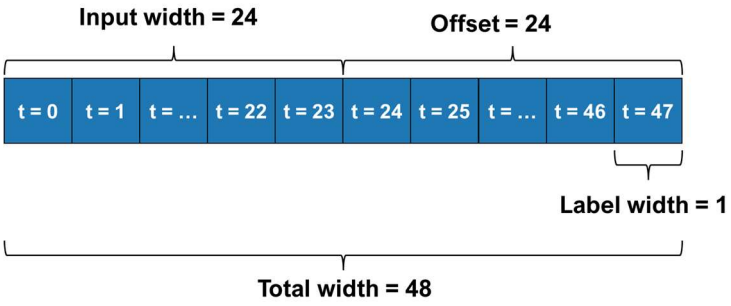

Figure 4: Example data window for making a single prediction 24 hours into the future, given 24 hours of historical data.

Feature Selection

For load forecasting, the following features were used:

- PrealL1
- PrealL2
- PrealL3
- EMS_total_load
- Day sin
- Day cos
- Week sin
- Week cos
- Year sin
- Year cos

For PV generation forecasting, the following features were used:

- AirTemp,
- CloudOpacity
- Dhi
- Dni
- Ebh
- Ghi
- GtiFixedTilt
- GtiTracking
- AlbedoDaily
- Ppv1
- Day sin
- Day cos
- Week sin
- Week cos
- Year sin
- Year cos

Single-step Output Models

The simplest model for time series forecasting predicts a single feature's value one time step (one hour) into the future based only on the current conditions. Here, our variable to predict is the total active power demand on the network (EMS_total_load), given the current value of all features which also includes EMS_total_load.

Baseline

Before building more complex models, it is desirable to have a performance baseline for reference. For our initial baseline, we start with a model that just returns the current demand as the prediction. Figure 5 illustrates the single-step predictions for the total load, made by a baseline model run over 24 hours. The blue `**Inputs**` line shows the input demand at each time step. The model receives all features, this plot only shows the demand. The green `**Labels**` dots show the target prediction value. These dots are shown at the prediction time, not the input time. That is why the range of labels is shifted 1 step relative to the inputs. The orange `**Predictions**` crosses are the model's predictions for each output time step. If the model predicted perfectly, the predictions would land directly on the `Labels`. The baseline predictions simply return the value of the previous step which resembles the original labels shifted one timestep to the right.

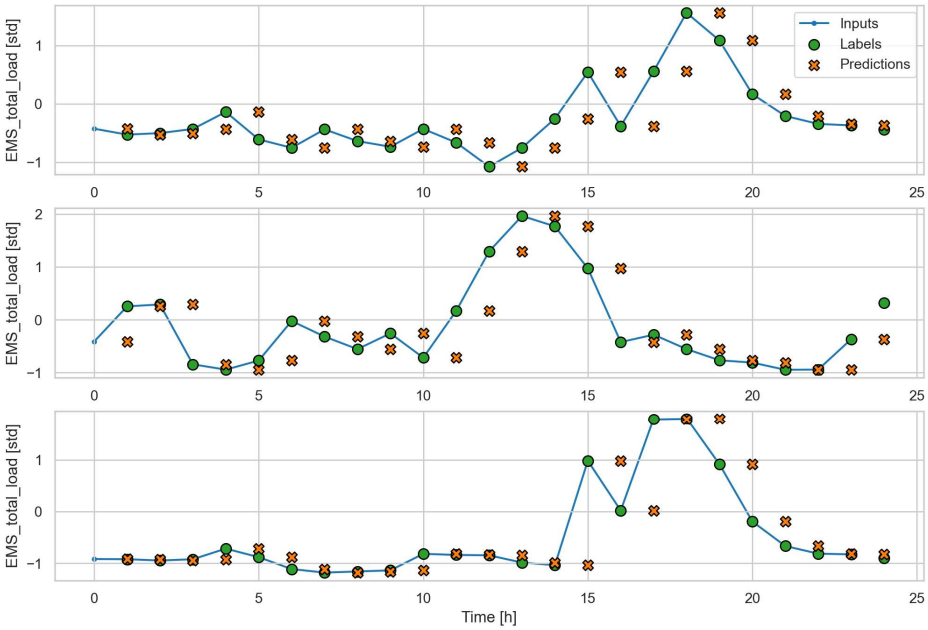

Figure 5: Example baseline predictions for a single-step output model.

Linear

The simplest trainable model that can be applied to this task includes a linear transformation between the input and output. As illustrated in Figure 6, the linear model contains a single node that takes all the features as inputs and assigns a weight to each input node which is essentially a multiple regression.

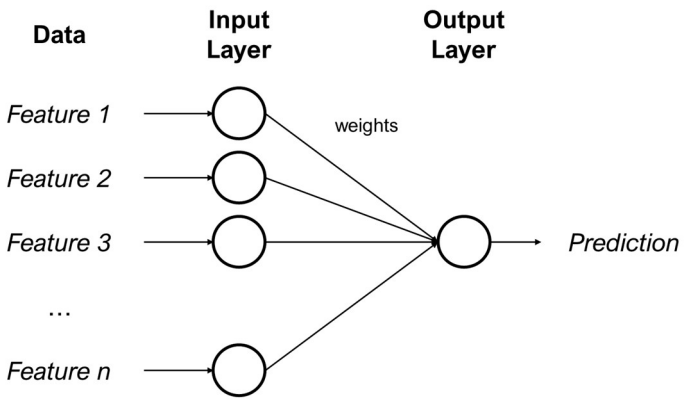

Figure 6: Linear single-step model architecture.

Dense

The dense model is similar to the linear model except it contains two dense hidden layers, each with 64 nodes, in-between the input and output layers. The architecture for the dense model is shown in Figure 7. In this case, the size of the dense layers is chosen arbitrarily and could be optimised via hyperparameter optimisation.

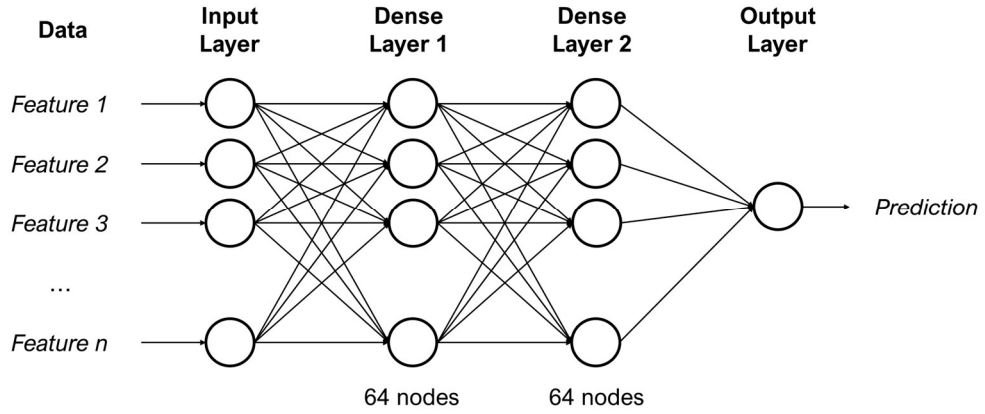

Figure 7: Dense single-step model architecture.

Multi-input Dense

The baseline, linear and dense models take in the data at each timestep individually. Here, the dense model is modified to take multiple timesteps as an input and produce a single output, shown in Figure 8. This is achieved using a ***tf.keras.layers.Flatten*** layer as the first layer of the model which modifies the shape of the input data from (timesteps, features) to (timesteps x features). The output is then reshaped as the last layer of the model using a ***tf.keras.layers.Reshape*** layer.

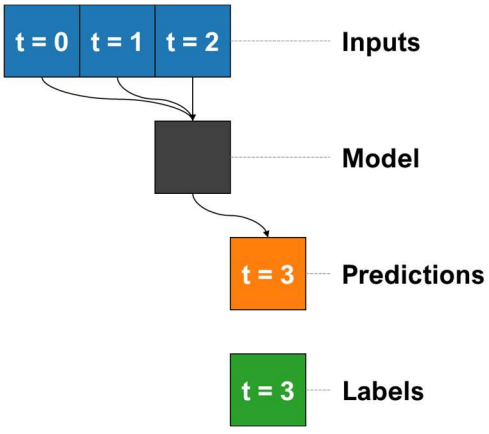

Figure 8: Multi-input Dense single-step model inputs and outputs.

Convolutional Neural Network (CNN)

CNNs are commonly used to analyse images [9], [10] and are characterised by their use of a convolutional layer which is also called a kernel or filter. A convolutional layer ***tf.keras.layers.Conv1D*** can also take multiple timesteps as input to each prediction. However, unlike the multi-input dense model, the CNN model can accept an input of any length, as highlighted in Figure 9.

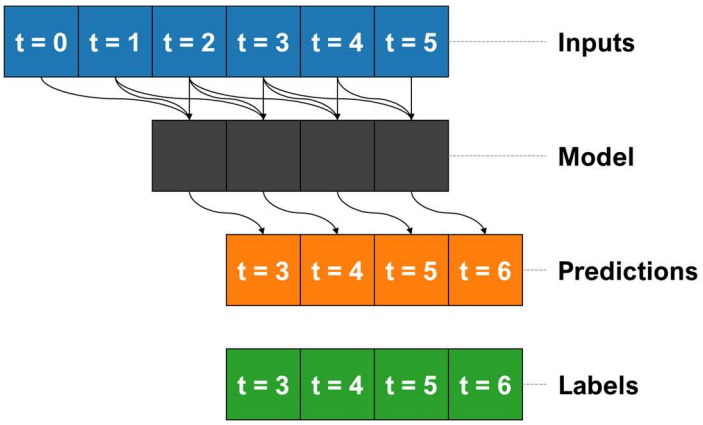

Figure 9: CNN single-step model inputs and outputs.

Long-Short Term Memory (LSTM) Network

A Recurrent Neural Network (RNN) is a type of neural network well-suited to time series data [11]. RNNs process a time series step-by-step, maintaining an internal state from timestep to timestep, as seen in Figure 10. Here, we introduce an RNN layer called Long Short-Term Memory (LSTM) ***tf.keras.layers.LSTM*** [12]. LSTMs were developed to deal with the vanishing/exploding gradient problem, experienced by regular RNNs, where the backpropagation error signal becomes vanishingly small or grows incredibly large [13].

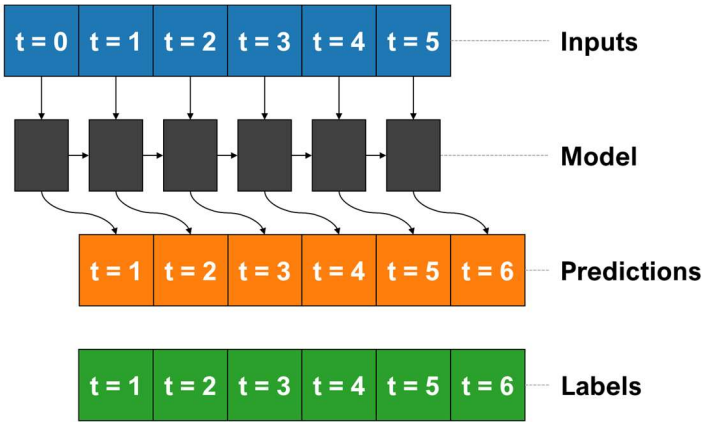

Figure 10: LSTM single-step model inputs and outputs.

Performance Comparison of Single-step Output Models

Figure 11 and Figure 12 compare the predictive performance of the single-step models on both validation and test datasets for electrical load and PV generation respectively. Mean absolute error (MAE) is used to evaluate the accuracy of the model's predictions and is calculated using

$$MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} \tag{2}$$

where y_i is the prediction, x_i is the true value (label) and n is the total number of predicted points.

Surprisingly, in both cases, the linear model performs worse than the baseline. All other NN models exhibit similar predictive performance and are all better than the baseline. Overall, the CNN model is marginally better than the rest, based on the test MAE.

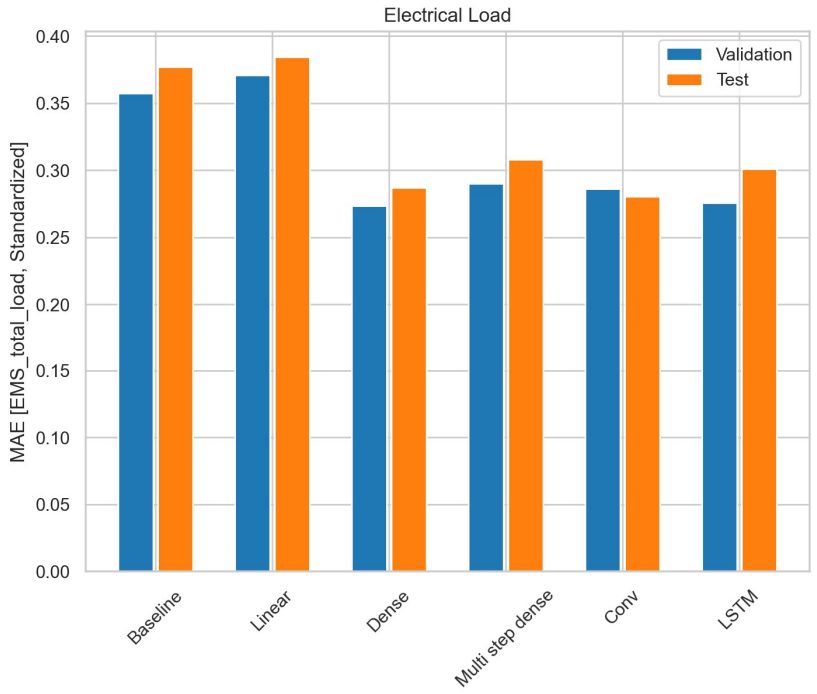

Figure 11: Comparison of single-step electrical load model predictive performance on validation and test datasets.

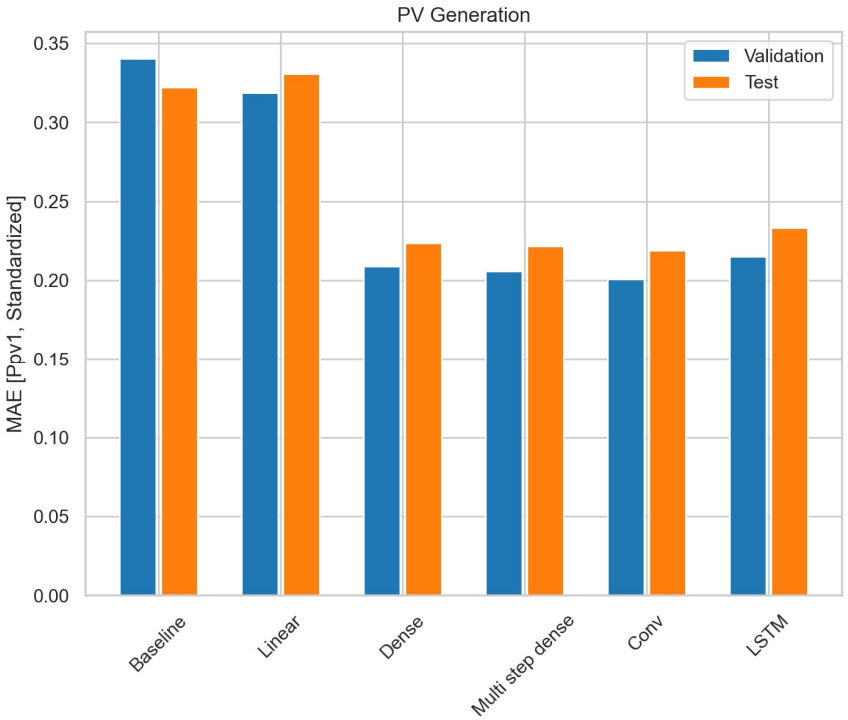

Figure 12: Comparison of single-step PV generation model predictive performance on validation and test datasets.

Multi-step Output Models

The single-output models in the previous sections made single time step predictions, one hour into the future. This section looks at how to expand these models to make multiple time step predictions 24 hours into the future, given 24 hours of the past. In a multi-step prediction, the model needs to learn to predict a range of future values. Thus, unlike a single-step model, where only a single future point is predicted, a multi-step model predicts a sequence of future values. Here, the multi-step models make predictions for all the features over all output time steps.

Baseline

First, we generate two baselines for prediction. The first repeats the last input value for the following 24 hours. The second repeats the entire 24-hour input window as the prediction.

Linear

The multi-step linear model behaves very similarly to the single-step linear model but here, it predicts the entire sequence in one go, as shown in Figure 13**Error! Reference source not found.**. This linear model should perform better than both baseline predictions but since the model needs to predict 24 values for demand from a single input time step with a linear projection, it is underpowered. This model is likely heavily reliant on periodic data such as the time of day.

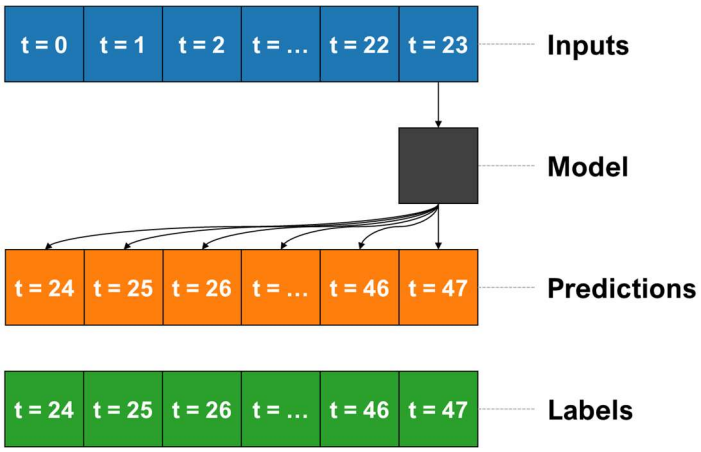

Figure 13: Linear multi-step model inputs and outputs.

Dense

The predictive accuracy of the linear model can be improved by adding dense layers between the input and output layers. However, this model still only uses a single time step to make a prediction.

CNN

The introduction of a convolutional layer allows a CNN to make predictions based on a fixed-width history, which should improve performance compared to the linear and dense models since it can observe changes to the features over time. The high-level model architecture is illustrated in Figure 14

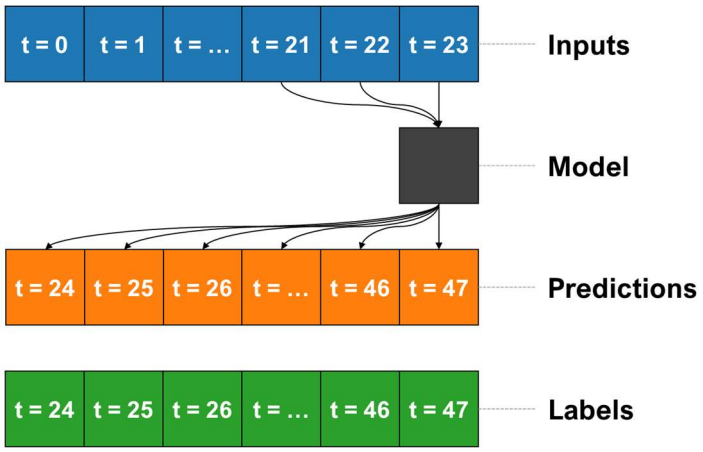

Figure 14: CNN multi-step model inputs and outputs.

RNN (LSTM)

An RNN can leverage a long history of inputs to calculate its predictions. Here, the LSTM model accumulates an internal state for 24 hours, before making a single prediction for the next 24 hours, shown in Figure 15.

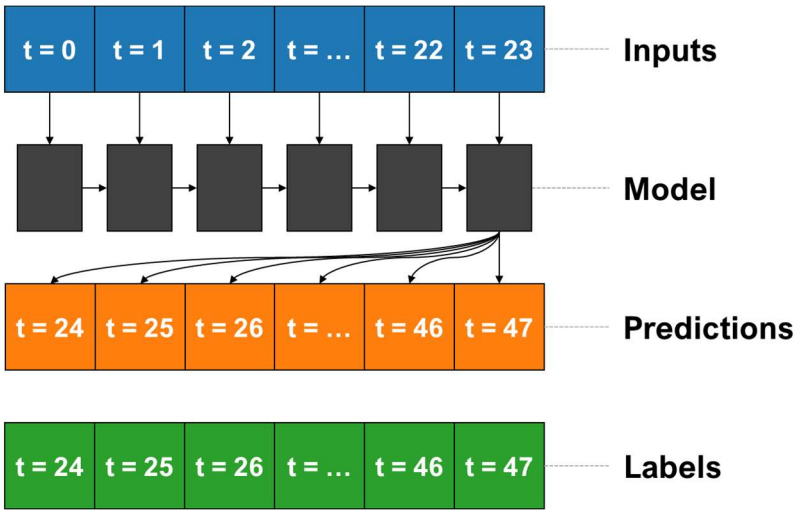

Figure 15: RNN (LSTM) multi-step model inputs and outputs.

Performance Comparison of Multi-step Output Models

Figure 16 and Figure 17 compare the predictive performance of the multi-step models on both validation and test datasets for electrical load and PV generation respectively. In both use-cases, using the last recorded value to predict the next 24 hours resulted in the largest MAE. Repeating the previous 24 hours performed unreasonably well. When predicting electrical load, only the LSTM model performed better than the repeat-baseline. For the PV generation predictions, the repeat-baseline performed the best. The next best model was the LSTM model. In both cases, the linear model performed the worst out of all the other NN models.

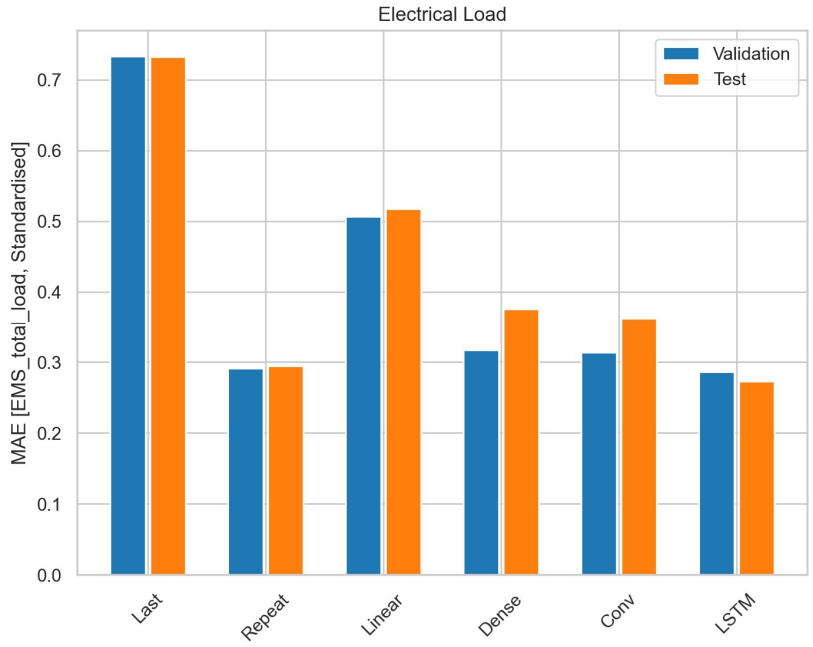

Figure 16: Comparison of multi-step electrical load model predictive performance on validation and test datasets.

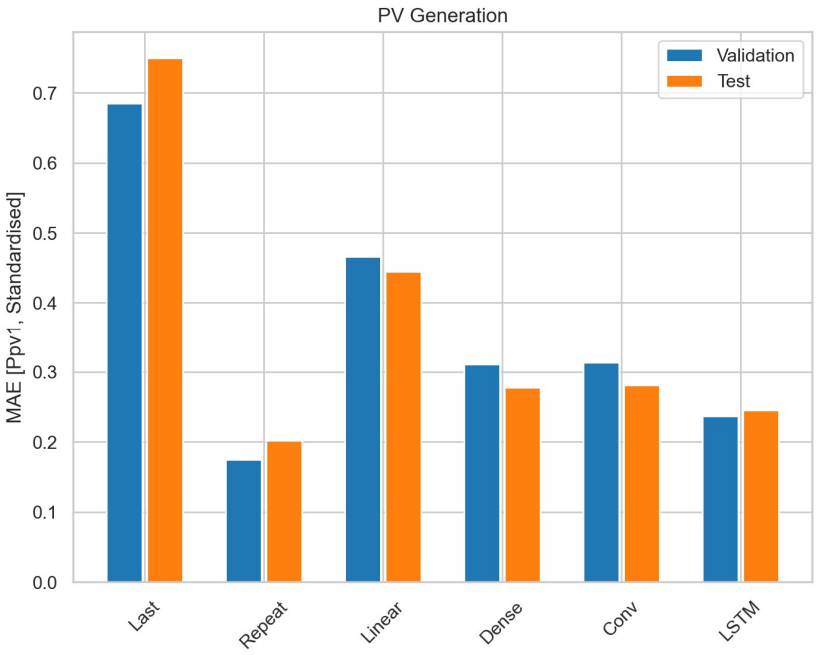

Figure 17: Comparison of multi-step PV generation model predictive performance on validation and test datasets.

5. Structural Time Series (STS) Models

Structural time series (STS) models are generalized additive models (GAMs), where the time series is decomposed into functions that add together to reproduce the original series [14]. An STS model expresses an observed time series as the sum of simpler components:

$$f(t) = f_1(t) + f_1(t) + \cdots f_n(t) + \varepsilon \tag{3}$$

where ε is gaussian noise. This can also be thought of as:

$$\text{observation}(t) = \text{trend}(t) + \text{seasonality}(t) + \text{impact\_effects}(t) + \text{noise} \tag{4}$$

One of the main advantages of STS is that they can often produce reasonable forecasts from relatively little data. In addition, the model's assumptions are interpretable, and we can interpret the predictions by visualizing the decompositions of past data and future forecasts into structural components. Moreover, STS models use a probabilistic formulation that can naturally handle missing data and provide a principled quantification of uncertainty [6].

Data Selection

Here, we will build STS models that try to predict the last week of electrical load and PV generation, given the previous four weeks of historical data. Figure 18 and Figure 19 highlight the training and test portions of the data for electrical load and PV generation, respectfully.

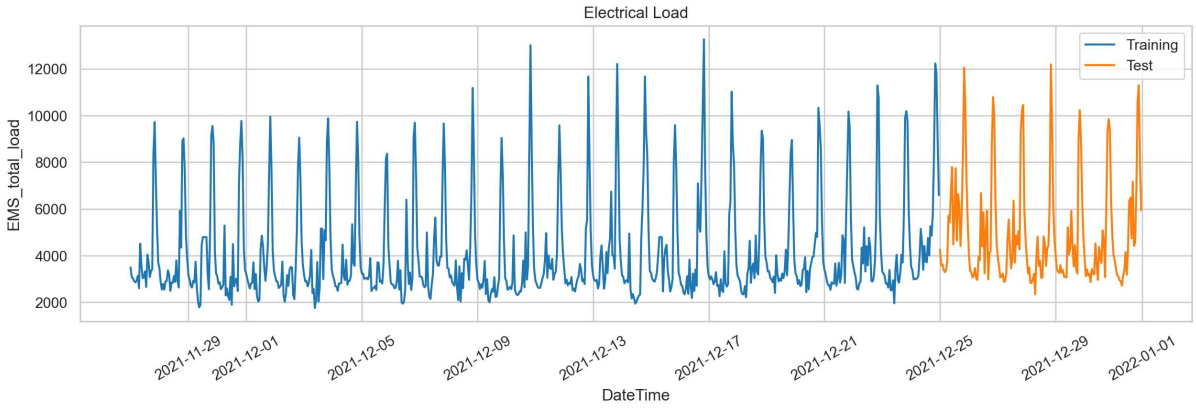

Figure 18: Training and test data split for the electrical load STS model.

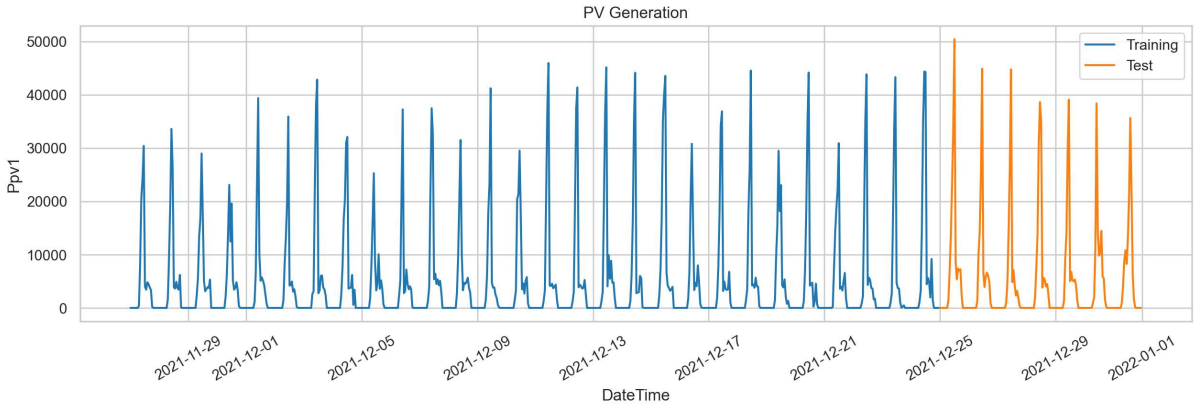

Figure 19: Training and test data split for the PV generation STS model.

Model Components

The models are defined as a sum of various seasonal and impacting effects. It is assumed that there is no significant trend effect as the data appears to be stationary over the five-week window. For the electrical load STS model the following additional features are assumed to contribute to the observed load:

- Hour of Day
- Day of Week
- Temperature

For the PV generation STS model, the following additional features are assumed to contribute to the observed generation:

- Hour of Day
- GHI
- EBH
- DNI
- DHI
- Cloud Opacity

Model Fitting

The models are fitted using variational inference (VI) in which an optimiser minimises a variational loss function called the negative evidence lower bound (ELBO) to approximate posterior distributions for the parameters. VI is a powerful method for approximating the posterior distribution in Bayesian models and has several advantages over other methods such as Markov Chain Monte Carlo (MCMC). VI is typically faster than MCMC which involves random sampling from the posterior distribution. However, in some cases, VI will produce a poor approximation of the posterior distribution when compared to MCMC. In TensorFlow Probability, this process is made easy using built-in functions ***build_factored_surrogate_posterior()*** and ***fit_surrogate_posterior()***.

Forecasts

After building and fitting the model, the built-in ***forecast()*** function can be used to make forecasts. The function takes the following arguments:

- *model*
- *observed_time_series:* This is the training time series data.
- *parameter_samples:* These are samples from the posterior distribution, estimated using VI.
- *num_steps_forecast:* The number of time steps you wish to forecast.

Figure 20 and Figure 21 show the week-long forecasts made by the electrical load and PV generation STS models, respectfully. The dotted orange line represents the mean forecast, whilst the shaded orange region represents two standard deviations.

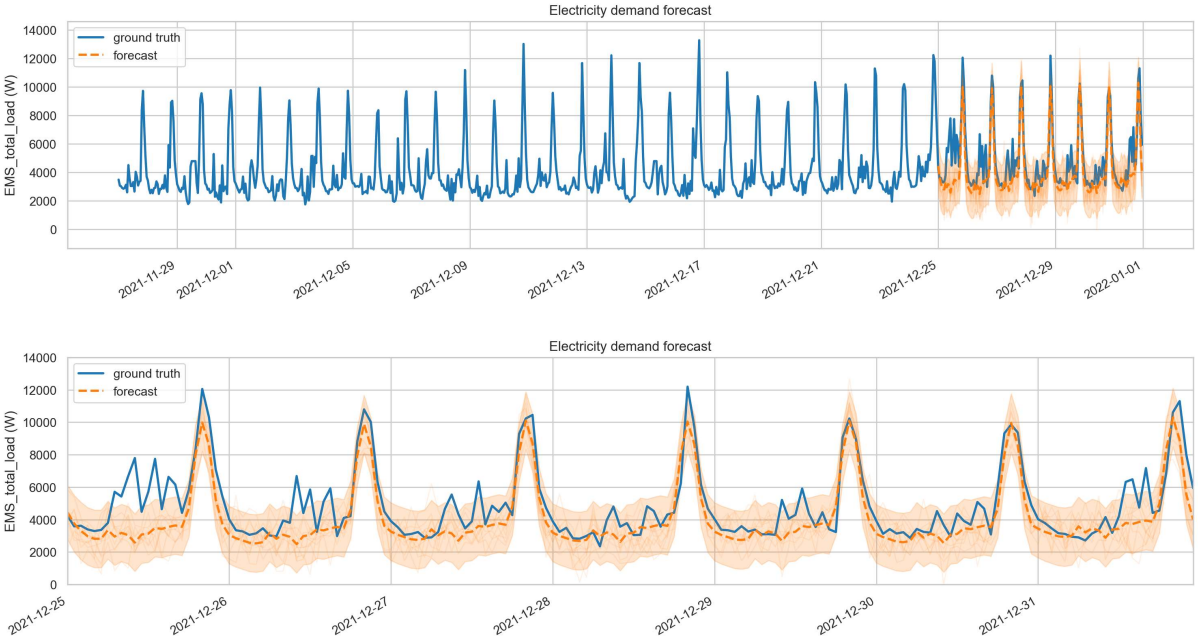

Figure 20: Electrical load STS model forecasts for one week, given four weeks of historical data.

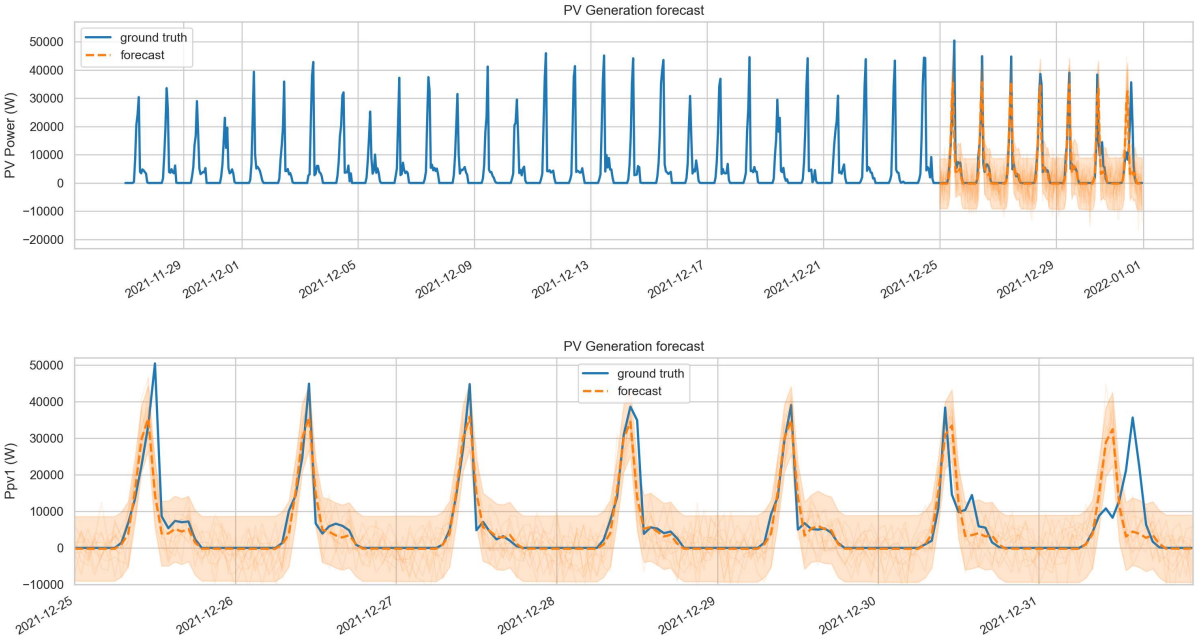

Figure 21: PV Generation STS model forecasts for one week, given four weeks of historical data.

Model Decomposition

With TensorFlow Probability, it is very easy to decompose an STS model using the built-in ***decompose_by_component()*** function. Figure 22 and Figure 23 show a breakdown of the seasonal and impacting effects for the electrical load and PV generation STS models, respectively. The shaded orange region represents two standard deviations.

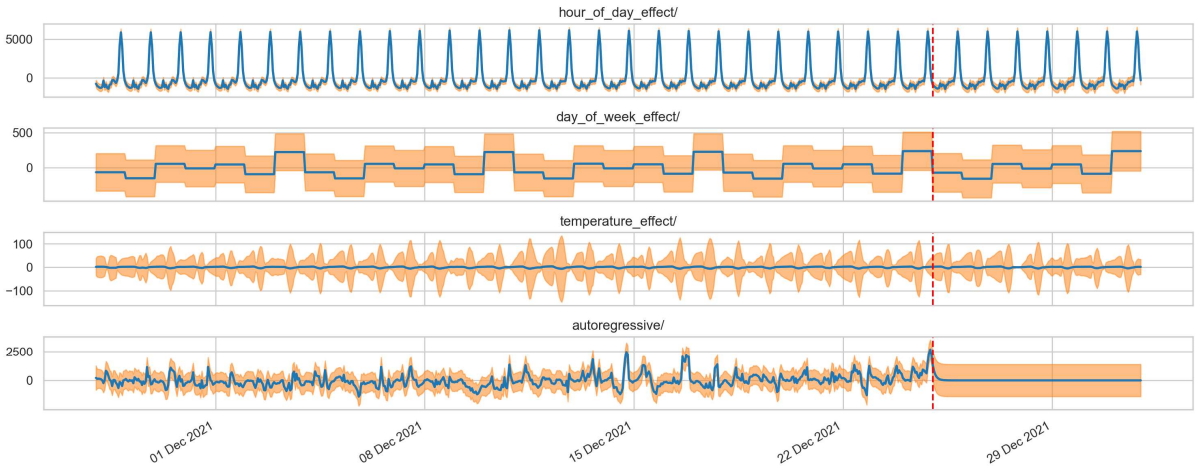

Figure 22: Seasonal and impacting effects for the electrical load STS model.

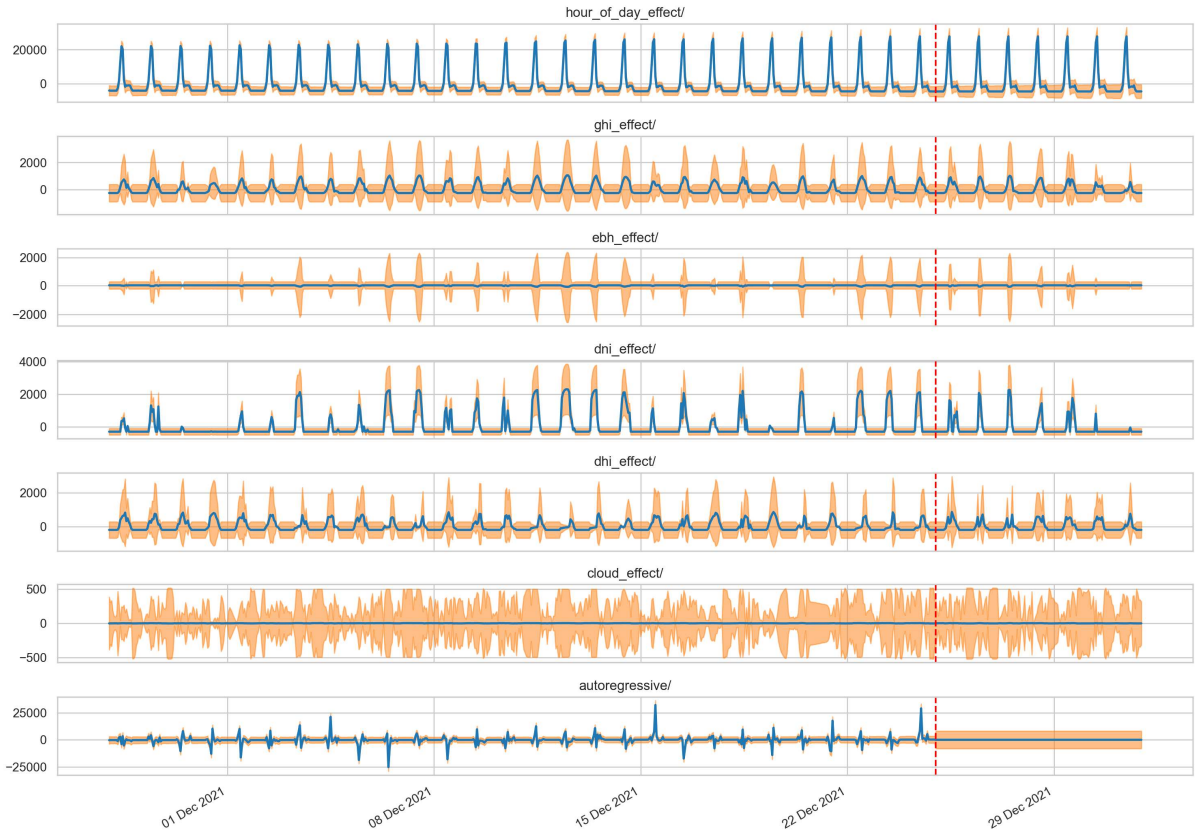

Figure 23: Seasonal and impacting effects for the PV generation STS model

Single-Step Forecasts

Single-step predictions can be made using the built-in function called ***one_step_predictive()*** which behaves in a similar way to the ***forecast()*** function, only without the number of forecast steps. Figure 24 and Figure 25 illustrate the single-step predictions made by the STS models across the five-week data window. The red crosses represent anomalies that are greater than three standard deviations from the mean prediction. Interestingly, the model appears to consistently underestimate the height of the mean PV generation in Figure 25.

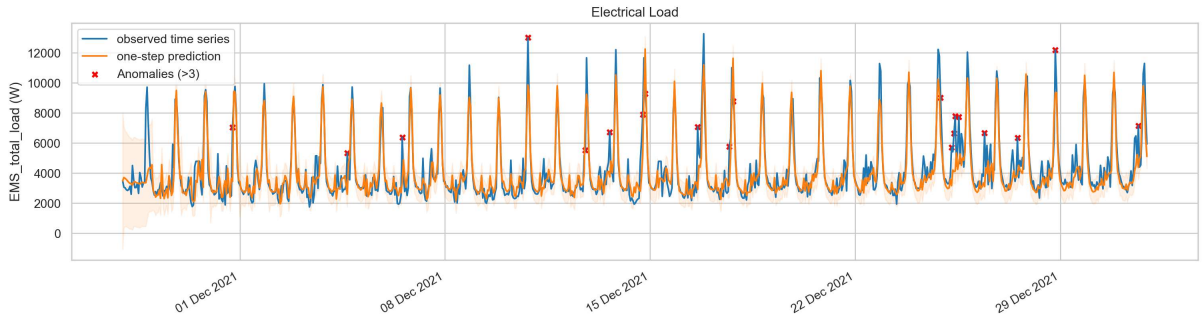

Figure 24: Single-step forecasts from the electrical load STS model for the five-week data window.

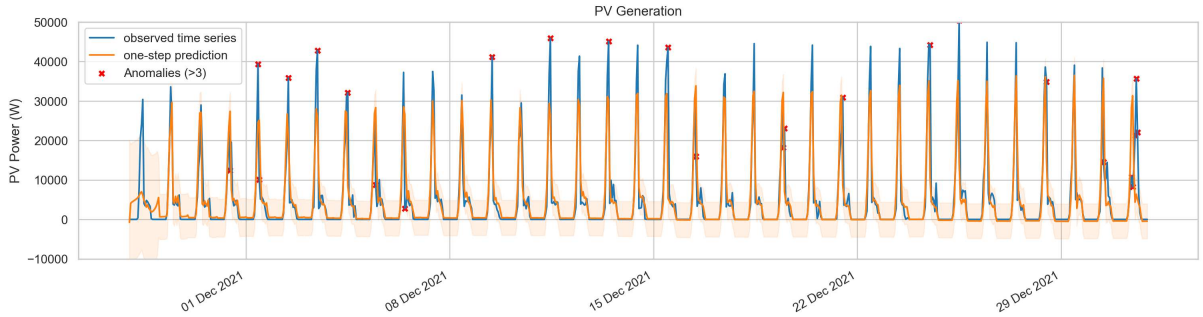

Figure 25: Single-step forecasts from the PV generation STS model for the five-week data window.

Model Evaluation and Comparison with Neural Networks

Figure 26 and Figure 27 compare the week-long predictions made by the STS model and LSTM NN model. Both models make very similar predictions, but the added uncertainty included in the STS prediction improves confidence in the model. The STS model can be used to quantify the probability that electrical load or PV generation will exceed certain values at a given time.

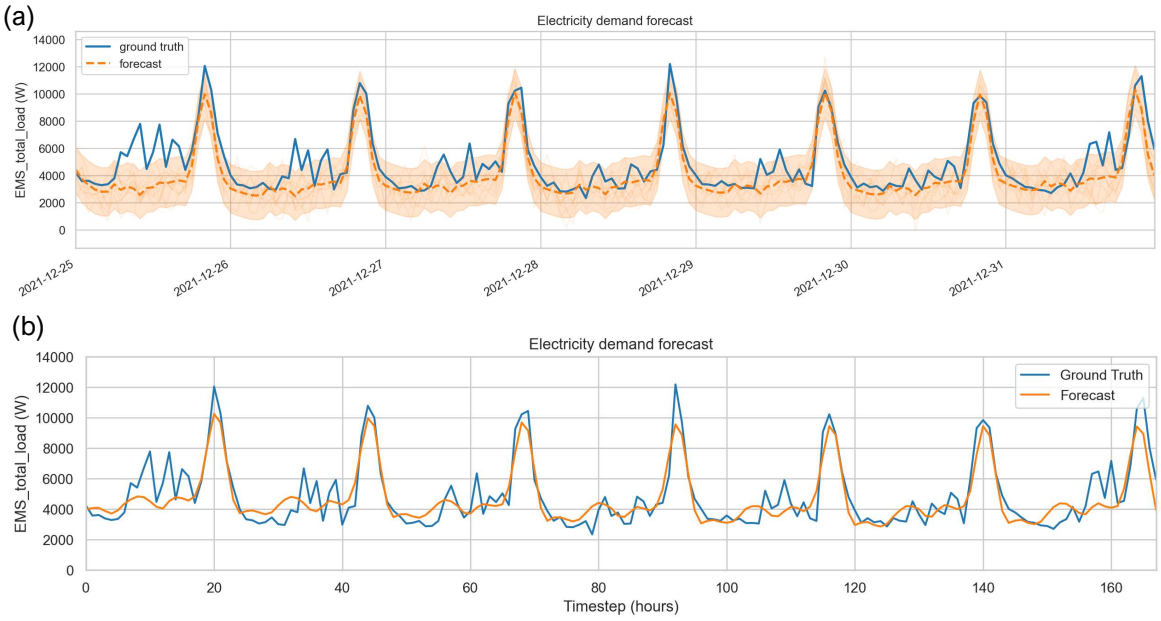

Figure 26: A comparison between the week-long electrical load forecasts made by the (a) STS model and (b) LSTM NN model.

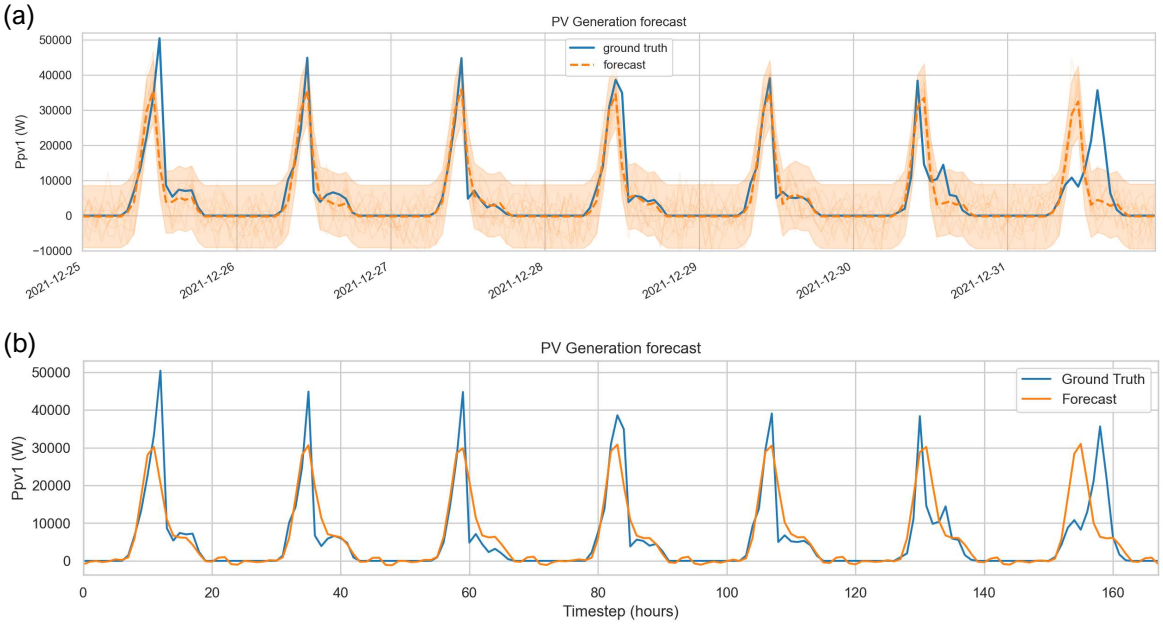

Figure 27: A comparison between the week-long PV generation forecasts made by the (a) STS model and (b) LSTM NN model.

Table 2: A comparison of MAE between the STS and LSTM NN models.

	Mean Absolute Error (MAE) in Watts	
Variable of Interest	**STS**	**LSTM NN**
Electrical Load	976	786
PV Generation	2389	2690

Table 2 shows a comparison of the week-long MAE scores for both models when applied to electrical load and PV generation forecasting. In this case, the STS model makes better mean predictions for PV generation, but worse predictions for electrical load forecasting.

6. Concluding Remarks

This report explores the use of NN and STS models for forecasting electrical load and PV generation. NN models can be used to model non-linear and complex relationships between variables in time series data, such as seasonality and trends. They are capable of learning from large datasets and can handle missing data, making them well-suited for time series data that is noisy or incomplete. Once trained, NNs can make accurate predictions on new data, particularly for short-term forecasting. However, NN models can be prone to overfitting when the training dataset is small or the model is too complex, leading to poor performance on new data. In addition, they can be computationally expensive and require large amounts of training data to achieve good performance. Moreover, the lack of explicit uncertainty estimates in the forecasts makes it difficult to assess the reliability of the predictions.

On the other hand, STS models capture the underlying components of time series data, such as trends, seasonality, and other cyclical patterns. They provide probabilistic forecasts that quantify the uncertainty in the model parameters, which improves trust in the model and can be useful for decision-making. Unlike NN models, STS models do not often require large amounts of data to make accurate predictions, making them suited to applications with small data. However, STS models may not perform as well as neural networks for highly non-linear time series data or when many variables are involved. They require careful specification of the model parameters and prior distributions, which can be difficult for non-experts. They require careful specification of the model parameters and prior distributions, which can be difficult for non-experts. They may be less suitable for short-term forecasting, as they capture long-term trends and underlying components of the time series data.

In summary, the choice between NN and STS models for time series forecasting depends on the data characteristics and the forecasting goals. NN models may be more suitable for short-term forecasting of non-linear and complex time series data, whilst STS models may be more suited for capturing long-term trends and underlying components of the time series data.

7. References

[1] C. Kuster, Y. Rezgui, and M. Mourshed, “Electrical load forecasting models: A critical systematic review,” *Sustain. Cities Soc.*, vol. 35, no. July, pp. 257–270, 2017, doi: 10.1016/j.scs.2017.08.009.

[2] A. Groß, A. Lenders, F. Schwenker, D. A. Braun, and D. Fischer, “Comparison of short-term electrical load forecasting methods for different building types,” *Energy Informatics*, vol. 4, no. Suppl 3, 2021, doi: 10.1186/s42162-021-00172-6.

[3] N. Ahmad, Y. Ghadi, M. Adnan, and M. Ali, “Load Forecasting Techniques for Power System: Research Challenges and Survey,” *IEEE Access*, vol. 10, no. July, pp. 71054–71090, 2022, doi:

10.1109/ACCESS.2022.3187839.

[4] B. Yildiz, J. I. Bilbao, and A. B. Sproul, “A review and analysis of regression and machine learning models on commercial building electricity load forecasting,” *Renew. Sustain. Energy Rev.*, vol. 73, no. December 2016, pp. 1104–1122, 2017, doi: 10.1016/j.rser.2017.02.023.

[5] TensorFlow, “Time Series Forecasting,” 2022. https://www.tensorflow.org/tutorials/structured_data/time_series (accessed Jan. 25, 2023).

[6] TensorFlow, “Structural Time Series modeling in TensorFlow Probability,” 2019. https://blog.tensorflow.org/2019/03/structural-time-series-modeling-in.html (accessed Feb. 16, 2023).

[7] I. Flower, “Resilient Project: Timeseries Forecasting Workshop,” 2023. https://github.com/isaacflower/resilient_project_ts_forecasting.

[8] M. J. Azur, E. A. Stuar, C. Frangakis, and P. J. Leaf, “Multiple imputation by chained equations: what is it and how does it work?,” *Int J Methods Psychiatr Res.*, vol. 20, no. 1, pp. 40–49, 2011, doi: 10.1002/mpr.329.

[9] N. Sharma, V. Jain, and A. Mishra, “An Analysis of Convolutional Neural Networks for Image Classification,” *Procedia Comput. Sci.*, vol. 132, no. Iccids, pp. 377–384, 2018, doi: 10.1016/j.procs.2018.05.198.

[10] B. B. Traore, B. Kamsu-Foguem, and F. Tangara, “Deep convolution neural network for image recognition,” *Ecol. Inform.*, vol. 48, no. October, pp. 257–268, 2018, doi: 10.1016/j.ecoinf.2018.10.002.

[11] J. T. Connor, R. D. Martin, and L. E. Atlas, “Recurrent Neural Networks and Robust Time Series Prediction,” *IEEE Trans. Neural Networks*, vol. 5, no. 2, pp. 240–254, 1994, doi: 10.1109/72.279188.

[12] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997, doi: 10.1162/neco.1997.9.8.1735.

[13] S. Squartini, A. Hussain, and F. Piazza, “Preprocessing based solution for the vanishing gradient problem in recurrent neural networks,” *Proc. - IEEE Int. Symp. Circuits Syst.*, vol. 5, pp. 713–716, 2003, doi: 10.1109/iscas.2003.1206412.

[14] Cloudera Fast Forward, “Structural Time Series,” 2020. https://structural-time-series.fastforwardlabs.com/ (accessed Feb. 17, 2023).