"Digital snow melt - Automated forecasting from snow parameters"

TUM Data Innovation Lab (TUM-DI-LAB)

Munich Data Science Institute (MDSI)

Technical University of Munich

&

ThinkOutside

Authors	Florian Donhauser, Fabienne Greier, Md. Forhad Hossain, Robin Mittas, Wudamu
Mentor	Dr. Juliane Sigl
Project lead	Dr. Ricardo Acevedo Cabra (MDSI)
Supervisor	Prof. Dr. Massimo Fornasier (Board of Directors of MDSI)

Mar 2022

Abstract

The forecast of seasonal snowmelt and inflow into water reservoirs based on automatic extraction of snow parameters plays a crucial role in the hydropower industry to plan and deliver energy more efficiently. This project aims to research appropriate Machine Learning (ML) methods for predicting the water inflow of specific reservoir lakes, implement a model based on an open-source ML model, train it with Norwegian data, and optimize the model prediction performance. To retrieve the Norwegian data, a `DataLoader` was created that downloads the weather station data from a website. In the downloaded data, many measurements were missing. Therefore, several techniques were developed to fill in the missing values. The target data of inflow into a water reservoir, provided by customers of the Norwegian startup ThinkOutside, had large fluctuations, so the temporal resolution had to be changed. After cleaning up and restructuring the existing code that served as a basis for the model, hyperparameter tuning was performed using Optuna to find the best configurations. Different prediction time frames were tested to evaluate the performance of the ML model. Overall, the accuracy of the predictions increased with tuned hyperparameters. However, some stations with noise in the year prior to the forecast showed poor performance. For these stations, it is recommended to use coarser methods for resampling the data.

Contents

1 Introduction

1.1 Motivation

In modern times, the demand for electricity is increasing rapidly worldwide. According to the World Energy Outlook 2021 by the International Energy Agency (IEA), the demand will be 30 % higher than the current consumption rates by 2030 [8]. However, fossil fuels, which have been the source of electricity for centuries, are running out. Even more critical than this depletion is the carbon footprint of electricity generation. Experts estimate that 60 % to 90 % of fossil fuels must remain unexploited to have a 50 % chance of keeping average warming below 1.5 °C [21].
For this reason, renewable energy sources with a smaller carbon footprint are becoming increasingly important. One primary energy source in this context is hydropower. The IEA reports in its special market report Hydropower 2021 that hydropower provides nearly 30 % of the world's capacity for flexible power supply and has the potential to contribute even more [7]. Snow is an important buffer for winter precipitation in this regard. It fills rivers and lakes when it melts in summer, contributing to hydroelectric generation. ThinkOutside, a Norwegian tech startup, contributes to this field by using radar technologies to provide information about snow. Predicting seasonal snowmelt with hydrological models can help hydropower companies plan and deliver energy more efficiently.

1.2 Problem definition and goals of the project

The goal of this project is to research appropriate Machine Learning (ML) methods for predicting the water inflow of specific reservoir lakes to provide planning security to customers. We want to provide an easily adjustable code-base to make these predictions, as some customers might need short-term predictions of the water inflow, e.g., a one-week prediction, but others also need long-term predictions to see the yearly trend. For this reason, we created several configuration files where we can specify many parameters, such as how many weeks we want to forecast, how many weeks we take as an input of the model, and several ML hyperparameters.
Many settings and hyperparameters have an impact on the performance of the model. Tuning them manually can be a tedious and lengthy process. Therefore, we also want to find an automated way to find the best model parameters for a given configuration.
Additionally, we want to include further weather data, such as precipitation or air temperature, which is publicly available, to increase the available input to the model and potentially its performance.

1.3 State of the art approaches and algorithms in this particular area

This section gives an overview of the current machine learning methods for automatically extracting snow parameters and estimating seasonal water inflow in reservoirs.
Traditional approaches to predicting this type of inflow, such as classical statistical models [17, 5] limited to short-term predictions or methods based on physical processes [18], cannot maintain model accuracy when dealing with nonlinear patterns in the data. Machine learning algorithms can overcome this shortcoming and improve the accuracy of inflow predictions. However, many attempts fail to predict peak inflows in long-term forecasts accurately.
A new multi-step forecasting approach is proposed by [6] to improve long-term water supply and inflow forecast accuracy. This approach uses historical time series data of snow water equivalent (SWE), defined as the equivalent amount of water if the snow mass is completely melted, and inflow to the reservoir to train an encoder-decoder algorithm. A long short-term memory (LSTM) encoder-decoder model with 16 nodes per layer is used to predict inflow into

the reservoir for future time steps. The LSTM is a deep neural network method well suited for predicting time series for long-term forecasting with multiple input variables [14] without the need to input a predetermined observation window [12]. The main goal of this project is to adapt this model, which was trained on US-Data, apply it to the corresponding Norwegian data, and forecast water inflow.
The model's input is the weekly mean SWE of several snow stations from 1990 onward. To increase the performance of the model, the input data was artificially enriched using the sequential index method (ISM) [10]. The basic idea is to take a time window of, e.g., 20 weeks as an input vector and a time window of the following 15 weeks as a target vector. Then, these input-output windows are created by iterating over the training data. Assuming we have ten different snow stations, one input window can be considered as a matrix $M \in \mathbb{R}^{20\times 10}$. Moreover, the input matrix M and the output vector $v \in \mathbb{R}^{15\times 1}$ are normalized to values $[0, 1]$, so that the network considers all input features to a similar extent during learning, since in summer months SWE values are usually closer to 0, while in winter months SWE values increase rapidly. This leads to faster convergence, as normalized data can accelerate the learning process [19].
To further improve the accuracy of the prediction, the model fitting is repeated several times, each run being independent of the previous run, as the prediction of every single run may vary in its accuracy. Afterwards, the mean of the predictions over all runs is calculated, resulting in improved performance.
At this point, we would like to refer to Figure 1, which shows U.S. inflow data, the target data of their model, compared to the Norwegian inflow data for one specific reservoir lake. Both data sets provide weekly values for the time series. The data presented in the plots is independently scaled to values in $[0, 1]$ as the size of the reservoir lakes differ and so does the inflow. It can be seen that the Norwegian data contains more noise, which makes it more difficult for the model to make accurate predictions. The model introduced by [6] achieved great performance but with very smooth inflow data. Let us especially remember the reservoir Einunna, and it's heavy drop around July 2020, the original inflow-value drops down to $-60\text{m}^3/s$ which of course is an error in the measurement, as the inflow can not be negative.
Consideration of other independent climate variables such as precipitation, air temperature, SWE, and snow thickness can potentially further improve accuracy. Therefore, as a second task in this project, other weather data are included, similar to the approach published by [16] in which local SWE and snowmelt timing are predicted using the LSTM method.

Figure 1: 2017-2019 Inflow Comparison US vs. Norwegian Data

2 Statistical data exploration

2.1 Data source

The first step was to acquire the data and to see where we can get which information from. For this project we are gathering the data from two different sources. The first part is downloaded from the open-source Norwegian Water Resources and Energy Directorate (Norwegian: Norges vassdrags- og energidirektorat or NVE). This data is provided by a governmental agency established in 1921 whose main task is to regulate the country's water resources and energy supply [22].
The second part is the water inflow, the target value of the model, which is provided by a third-party customer company. It is an excel file that contains over 40 years of data for several reservoir lakes of interest to the client. During the last week of the project, we also received a second file containing a time series starting in 1970 for one further lake.
Previous work [6] has shown that a trained deep neural network can predict inflow for future time steps using historical time series data for snow water equivalent (SWE) and reservoir inflow. Our data is also based primarily on these two features. Weather and SWE data can be downloaded from [9]. Here, we can select many different weather stations that can measure different features. The information about each selected station is illustrated in Figure 2, Table 1 and Table 2.

Figure 2: Station map

2.2 Data visualization

To visualize the downloaded data, we have selected the station „Anestølen“ since it can measure most of the features. The measurements for air temperature, snow water equivalent, precipitation, wind direction and speed, relative humidity, and snow depth are shown below. Figure 3 shows the time series for these features from 12-03-2021 to 12-10-2021.

2.3 Data acquisition

To download the data for different stations and features, we have created a table called "station information" which contains all the basic information about each available station. We have developed a `DataLoader` with different functionalities. Based on the selected stations and selected features, the `DataLoader` can download the data from the website [9] for the given configuration. The user can specify a start time, an end time as well as different time resolutions to download the data. As some stations send signals every couple of hours and others once a day, we have decided to download a daily time series.

Table 1: Information about the station for collecting data.

Name	ID	Latitude	Longitude	Service since	Measurements index
Groset	16.232.14	59.836	8.314	October 2000	2
Øvre Leirbotn	213.7.0	70.128	23.617	January 1999	1,2
Grimsa snøstasjon	2.373.0	62.063	9.991	October 2007	1,2
Siccajavvre	212.23.0	68.755	23.538	October 1997	1,2
Storstilla ndf. Balvatn	164.12.0	67.013	15.973	October 1997	1,2
Namsvatn tunnel Vekteren	139.4.0	64.961	13.540	October 1997	1,2
Maurhaugen - Oppdal	121.2.0	62.660	9.820	October 1998	1,2
Brunkollen klima	8.5.0	59.968	10.547	November 1983	1,2,7
Øverbygd rør 2	196.47.2	69.022	19.354	July 2004	1,2,8
Bakko	12.142.0	60.681	8.004	October 1998	1,2,7
Anestølen	77.24.0	61.366	6.906	September 2011	1,2,3,4,5,6,7
Kyrkjestølane	73.11.0	61.178	8.113	October 1967	1,2,3,4,5,6,7
Vetlebotn	62.42.0	60.866	6.473	September 2016	1,2,4,5,6,7
Svarttjønnbekken Klima	123.93.0	63.319	10.648	January 1970	1,2,3,5,6
Nedre Sjodalsvatn	2.13.0	61.560	8.927	January 1930	1,9,10,11
Øvre Heimdalsvatn	2.36.0	61.418	8.895	June 1995	1,2,3,4,6,7,9,10

Table 2: The measurable features.

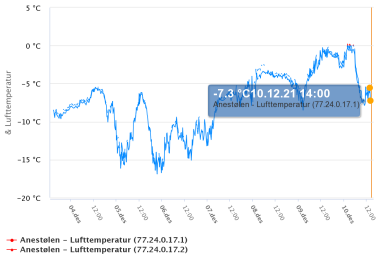
Measurements	Index
Air temperature	1
Snow water equivalent (SWE)	2
Precipitation	3
Wind direction and speed	4
Relative humidity	5
Wind speed	6
Snow depth	7
Groundwater level	8
Discharge	9
Stage	10
Reservoir volume	11

Once the Python scripts to download and prepare the data have run successfully, the input and the inflow data are merged in `ResCNN_LSTM/ResCNN_LSTM_fit.py` (Appendix B), and the data is converted to, for example, weekly averages, depending on the selected station and configuration.

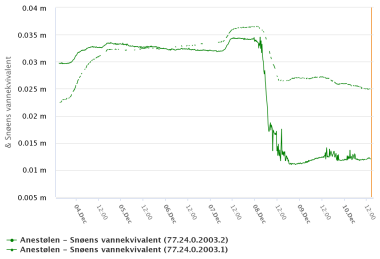
2.4 Data cleaning

Due to the different construction times of the stations, repair-related down-times, or other circumstances, some stations cannot send signals, or the data is generally not available. Therefore, many values are missing in the downloaded data. Hence, we have developed a data cleaning and pre-processing tool to detect stations that are missing too much data and to fill in the missing values.

In addition, we have the provided reservoir inflow data, calculated as the sum of the inflow to the actual reservoir and the upstream reservoir. Since the inflow is often a small value compared to the outflow and the changes in reservoir volume, relatively small errors and inaccuracies in water level or outflow can result in large errors in the calculated inflow. Another explanation given by a customer for the jumps in the data is that the reservoir pressure sensor may be located too close to the intake of the power station. Consequently, the pressure sensor is affected by the

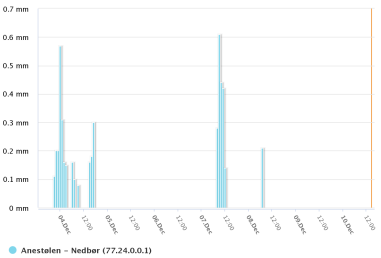


(a) Air temperature

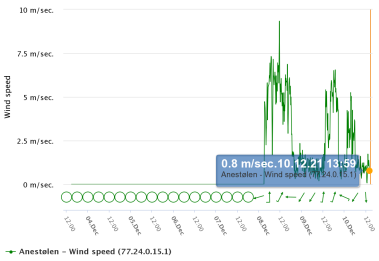


(b) Snow water equivalent

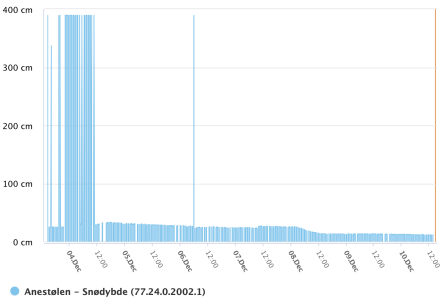
(c) Relative humidity



(d) Precipitation



(e) Wind speed direction



(f) Snow depth

Figure 3: Visualization of the measurements

power generator turning on and off.
Therefore, we do not use the daily inflow values directly as the fluctuations, as shown in figure 4, are too large to make accurate predictions. Depending on the reservoir lake and its data quality, we can specify different resampling methods in our configuration file. It might be sufficient to take weekly to three-weekly averages of the inflow to reflect annual trends.

2.4.1 Handling NA values of the input data: interpolation

After downloading the data, we prepare and process it to get the highest possible benefit. First of all, we drop columns with more than 55% of NA values, as these missing values cannot be interpolated accurately and could lead to incorrect data, which could affect the model's prediction. In our configuration file, we can specify the threshold for dropping a column.
Suppose a station was built one year after the start date of the input data. Thus, for the first year, the column consists only of Null values. For later years, there might be very few Null values. That is why we iterate over all years of the data as well. Linear interpolation makes sense for some years, such as the later one, while for a year for which we have no values at all,

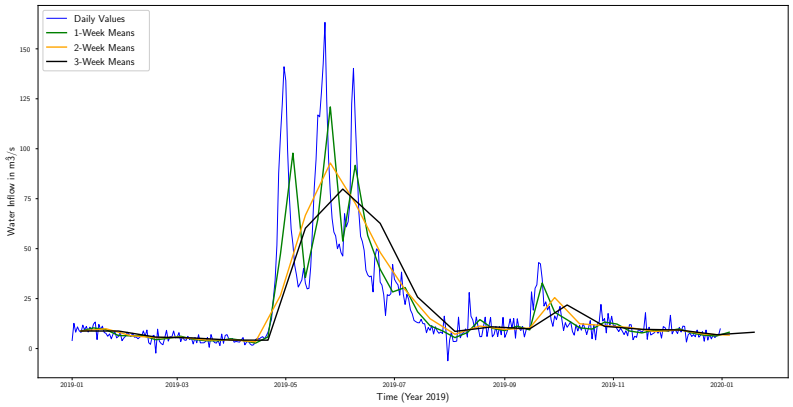


Figure 4: Inflow Values of Norwegian Reservoir Lake 'Aursunden' with different Resampling Methods.

it makes more sense to take the average over all other years to fill in the Null values.
We have also implemented a counter for the consecutive days with Null values of each station in each year. Imagine that a station only sends a signal every second day, then linear interpolation would be very useful even if the station contains 50% missing values. An example of this is shown in Figure 5. So we had to consider several different scenarios. Another good example of linear interpolation is shown in Figure 6. If a station has less than 55% of Null values for the observed year and at the same time has a maximum of consecutive days with Null values of approximately 60-90 days in this year, linear interpolation is a good fit for the downloaded data. In the case in which a station does not send signals for more than three consecutive months, we consider different methods. Depending on the specification set in the configuration file, different methods are executed to fill in the missing values.

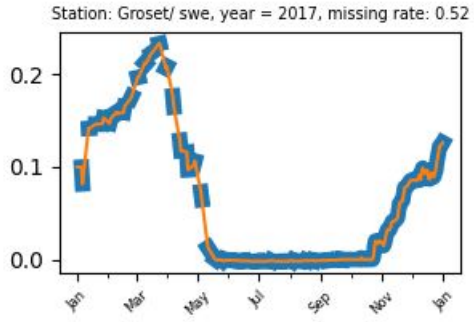
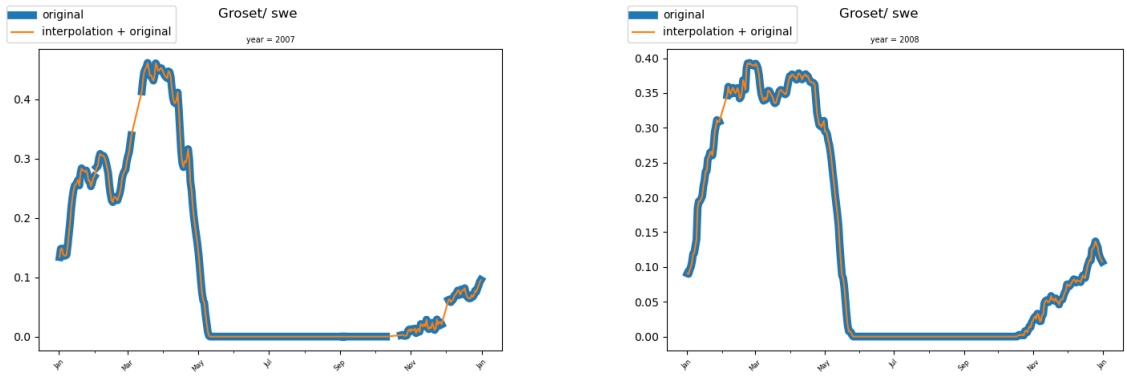


Figure 5: Blue: Original, Orange: Interpolated. Example for a year when linear interpolation is well suited even though it has a high missing rate

2.4.2 Handling NA values of the input data: averaging values of the station over all other years

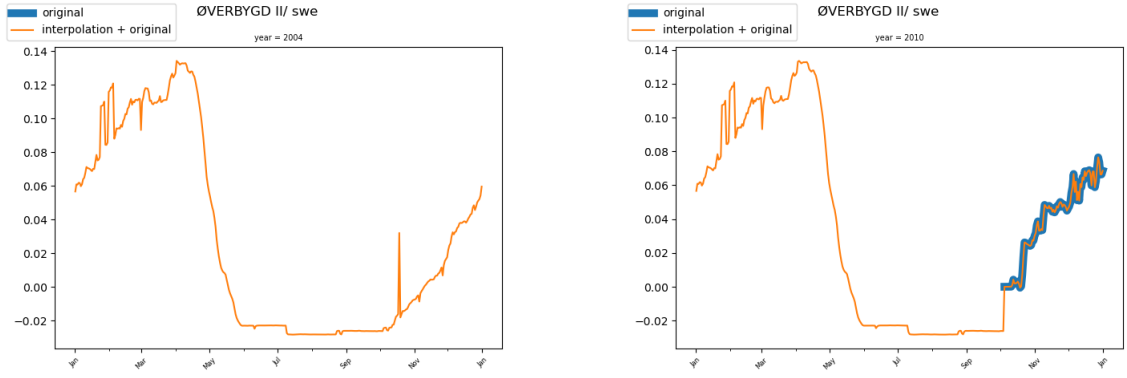
Another approach was to fill in the missing values with the average of the same station over all other years for which data is available. To this end, we calculated a time series of length 366,



(a) (b)

Figure 6: Visualization of Linear Interpolation

in which, for instance, the first element is the average of this station over all other years on the first day of the year. Figure 7(a) shows for the station „Øverbygd II (SWE)“ a year for which no data is available, so we filled in the missing values with the averages over all other years from this particular station. Figure 7(b) shows an additional example of this method where some data was available for the considered year with a maximum of consecutive Null Values greater than 90. This method is best suited for cases in which there is barely any data available.



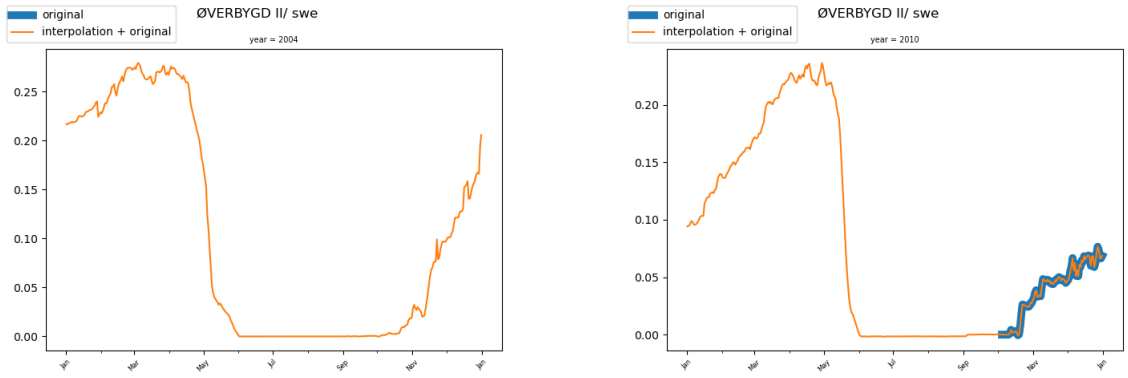
(a) (b)

Figure 7: Visualization of Average Feature from other years

2.4.3 Handling NA values of the input data: averaging values of the same day from same feature stations

We have also incorporated a function that calculates the averages for all columns with the same feature on a given day. Assuming we have a Null Value at 2005-01-01 for an SWE station, our function then calculates the average over all other SWE stations on the same date and fills the Null value with the average. Figure 8 shows an example of this method, for which we consider the same station and years as in figure 7. We can now see that this method obviously fills Null values differently than before. From January to September/October, the values differ from the previous method. Both approaches can be useful, with almost no difference in the accuracy of the model. However, we suggest using this method as it is more realistic to take the average

values from stations in the same region for a given day since they are most likely to have similar weather conditions on an observed day.



(a) (b)

Figure 8: Visualization of filling missing values with Averages from same feature stations

2.4.4 Time resolution of the data

As shown in Figure 4, the choice of an appropriate temporal resolution is crucial. For the inflow data, we need to use at least weekly averages of the data. However, the fluctuations are still too large for some stations to make accurate predictions. In this case, we can also choose a coarser time resolution of, e.g., two or three weeks.

3 Use cases

The relevance of time series predictions using an LSTM encoder-decoder architecture for the hydropower industry has already been discussed in Section 1.1. However, the results of seasonal snowmelt forecasts with accurate, timely snow data can also be extended for applications in different contexts like planning drinking water supplies or preventing flooding in residential areas.

In addition to snowmelt predictions, the developed model can be adapted to other areas where time series data, i.e., a collection of observations obtained through repeated measurements over time [3], is available. Best suited for the model is seasonal patterned observation data. The property of the LSTM model of having a cell state is beneficial for this type of data. The memory capability makes it possible to store information and retrieve patterns in the data over a long period of time. In this way, the model can make predictions for the future based on past observations.

Staying in the power sector, the developed model may be helpful to predict the output power of photovoltaic systems, which is essential to ensure reliable operation and economic integration of photovoltaics into smart grids [1]. Regarding climate change, it is possible to predict sea surface temperature using Argo data [15]. Ocean temperature variations have important implications for marine ecosystems as well as global climate change. The method can also be useful in preventing and predicting natural disasters to save human lives. Exploiting spatio-temporal correlations between earthquakes at different locations can improve the prediction of early warning systems [20].

Another application of LSTM models is stock price prediction, which can be used for the development of a trading strategy or deciding the right time to buy or sell stocks [11]. The proposed model could be adopted to this sector. Another use case related to smart cities is traffic flow forecasting based on air pollution and atmospheric parameters [2]. Accurate traffic forecasts are essential for traffic management, as they help transportation authorities manage traffic and help drivers choose the most efficient route to their destination.

The current spread of COVID-19, which is affecting our daily lives, is a recent example where LSTM can help make better predictions and assist policymakers in responding quickly to changes in the number of new infections [13]. Moreover, the predicted data can help guide action and make informed decisions with a positive real-world impact.

4 Model architecture

Our project is based on the code by Herbert et al. [6], as described in the Introduction. We use their results on the different model architectures and have decided to use and slightly modify their developed LSTM encoder-decoder architecture. To improve code quality, flexibility, and readability, we have decided to restructure their code and change how the model is defined in TensorFlow. Instead of defining the model with the TensorFlow functional API [4], we have decided to subclass `tensorflow.keras.Model` and to move the code for the model into separate files.
Furthermore, the model consists of similar or even identical parts which are repeated. The original code contained code duplicates for those parts. These reused elements consisting of multiple layers are often called blocks, and we are also using this terminology. In our case, the block consists of two convolutional layers with a skip connection, followed by a ReLU activation function. This is done twice, and finally, one-dimensional max-pooling is applied to the output. We call this block `ResCNN_Layers` and create it by subclassing `tensorflow.keras.layers.Layer`. Since the dimensionality of the output space for each block is not identical throughout the model, we implement it as a parameter for the constructor of the block, so it can be changed. The same applies to the kernel size of the convolutions, which is also a parameter. A figure for such a building block is shown in Figure 9.
Now that we have defined the main building block, the architecture is created by stacking three of these blocks together, forming the CNN part of the model. Afterwards, the output is flattened and repeated the same number of times as the number of values we want to predict. Next, the data is passed through 4 LSTM layers. Finally, the model has two dense layers wrapped in a `TimeDistributed` wrapper, ensuring that the dense layers are applied individually for each timestamp. The full architecture can be seen in Figure 9.
Since the whole model is defined in separate files and as a subclass of
`tensorflow.keras.Model`, we can easily create it in the main file in just one line of code. Additionally, all numbers for the kernel size, number of output values, number of filters, and number of neurons in the dense layers are passed in as arguments which themselves are read from the config file. This allows for easy modifications to the model architecture and also changes during the hyperparameter tuning to investigate the influence of these settings on the final prediction accuracy.
During the project, we have noticed that overfitting is present during the training process. Therefore, we have incorporated common techniques to combat overfitting for the model architectures. A dropout layer is added between the dense layers at the end of the model. Furthermore, the dropout probability is also used for the output of the LSTM layers and their hidden states. By setting the dropout probability to 0, the dropout can be disabled. Additionally, batch normalization layers can be added after the one-dimensional convolutions. There is a setting for adding these layers in the config file. Lastly, we are using L2 regularization, also called weight-decay, for the convolution and LSTM layers. The L2 regularization factor can be set by the user as well. Since all these techniques can be changed using the config file, we also include them in the hyperparameter tuning.

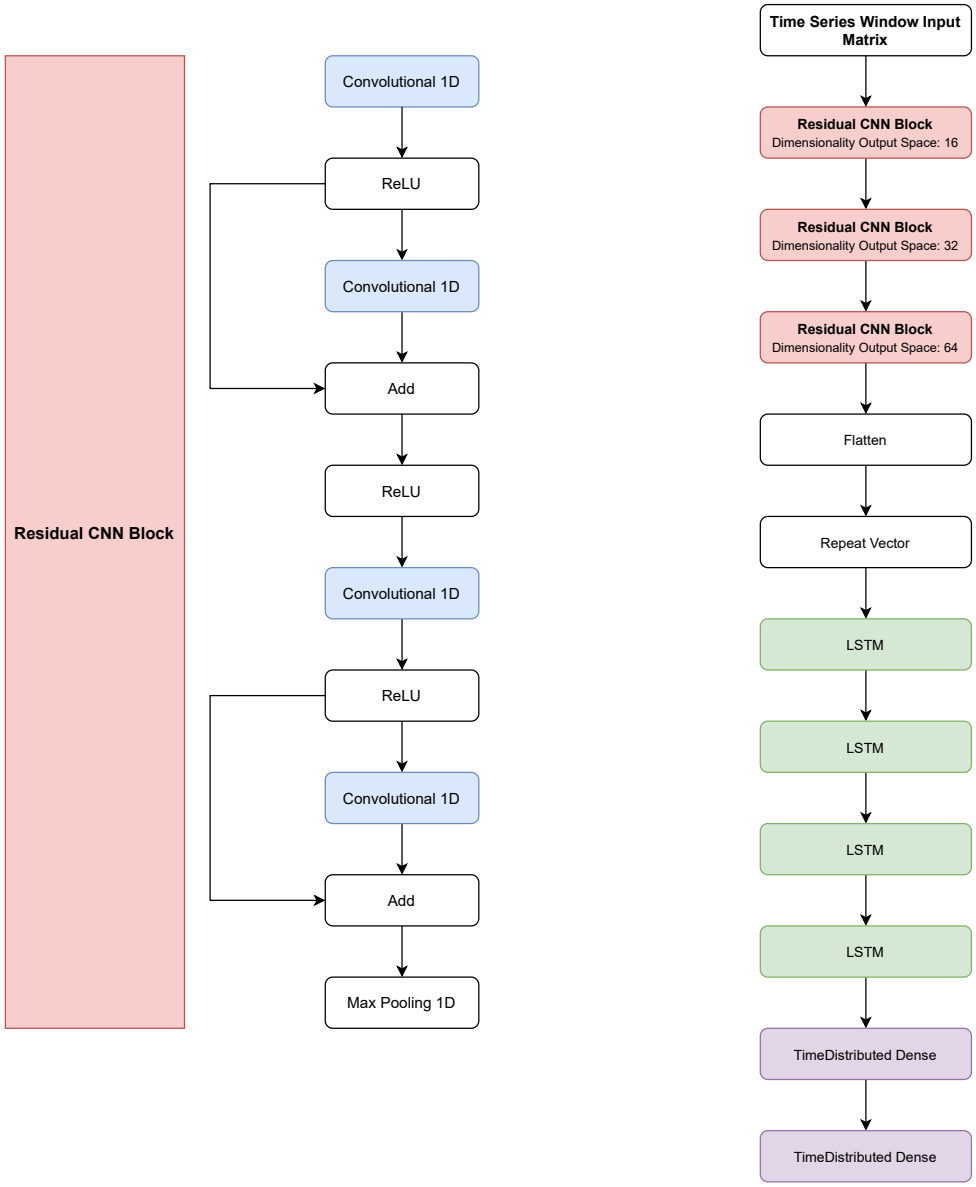


Figure 9: On the left side: Residual CNN Block. On the right side: Complete model Structure. Coloured Layers contain trainable parameters

5 Hyperparameter optimization

A Machine Learning or Deep Learning model usually consists of two sets of parameters. Firstly, model parameters or weights which are changed and set during the training process to fit the data, which are learned values. Additionally, other parameters need to be decided on by the developer and are not learned from the data, they can be understood as a specific configuration for the ML training. These parameters are generally called hyperparameters.
In our model, we are using the learning rate, L2 regularization factor, the batch size, the kernel size, the length of the input window, the number of epochs, the size and dimensionality of the different layers, the dropout probability, the batch normalization, and the patience for early stopping as hyperparameters. An explanation of all these hyperparameters which can be set in the config file can be found in the appendix.
Choosing the best set of hyperparameters is not an easy task. For example, if we choose a rather high learning rate, we might skip the point of global optimum. In contrast, if we choose a rather small learning rate, it will take a very long time to train the model. Moreover, the model may get stuck in some local minima due to the small learning rate.
There are two approaches we could follow for hyperparameters optimization. The first is manual hyperparameter tuning, and the second is automatic hyperparameter tuning using a framework. At first, we tried the manual approach, and then implemented the automatic one.

5.1 Manual hyperparameter optimization

Initially, we picked a tedious approach to optimize the hyperparameters. We trained the model several times with different combinations of hyperparameters, which we manually changed. Using this approach, we found the following table with different hyperparameters.

Epochs	Loss	Batch size	Optimizer	Repeats	MAE	RMSE
50	MSE	32	Adam	5	18.78	26,06
50	MSE	32	Adam	2	19.26	25.07
30	MSE	32	Adam	20	23.56	29,85
100	MSE	32	Adam	2	25.15	34.31
40	MSE	64	Adam	3	27.86	38.33
20	MSE	64	Adam	20	31.88	42.92
100	MSE	32	Adam	5	35,157	49,662
30	MSLE	64	Adam	3	51.92	68.99

Figure 10: Manual Hyperparameter Tuning

After creating this table, we compared different settings with respect to the validation error. We can see that 50 epochs with 32 batch sizes and 5 repetitions produce the least validation error, whereas 100 epochs with the same batch size and repetitions produce the largest validation error.
This approach has two problems: We have to make the combinations manually, and it is very likely, that we will not find the best set of hyperparameters as this process is relatively slow and few configurations can be tested.

5.2 Automatic hyperparameter optimization

Based on the manual hyperparameter optimization, we can see that the optimization or tuning of hyperparameters is tedious. However, the performance of the model may highly depend on the choice of the best set of hyperparameters and hyperparameter tuning is therefore important. Hence, we needed to find a technique to tune hyperparameters with less manual work, preferably automatically. There are several hyperparameter tuning frameworks: Optuna, Ray-Tune, Hyperopt, MLMachine, and Polyaxon are some major frameworks. We used Optuna for its easy parallelization, quick visualization, efficient optimization, eager search space, and lightweight, versatile and platform-agnostic architecture.
Hyperparameter optimization with Optuna is designed so that we can automatically run the model multiple times, each time with different hyperparameters. Each run is called a trial. The hyperparameters can be selected from different distributions. We use hyperparameters which are categorical, integers in a range, a float from a uniform distribution, or a float from a log-uniform distribution.
In our model, we have used more than twenty hyperparameters. Among those, we have used Optuna to tune thirteen hyperparameters. Below are the hyperparameters that we optimize using Optuna. The details of all configurations can be found in the Appendix.

```
n_timesteps : integer value, range between 8 and 102
batch_size : categorical value, from  [64, 128]
epochs : integer value, range between 25 and 100
n_nodes1 : integer value, range between 4 and 128
n_nodes2 : integer value, range between 4 and 64
filter1 : integer value, range between 2 and 128
filter2 : integer value, range between 2 and 64
kernel_size : integer value, range between 3 and 9
learning_rate : log uniform value, range between 1e-5 and 1e-3
dropout_probability : float value, range between 0 and 0.5
use_batch_normalization : categorical value, from [True, False]
patience : integer value, range between 5 and 50
weight_regularizer : log uniform value, range between 1e-9 and 1e-5
```

We did not pick these ranges or categorical values from the beginning. Initially, we chose a larger range. Then, we ran the Optuna several times for 200 trials. This helped us narrow down the ranges and the categorical values. The following graph shows the distribution of validation losses of 200 trials.

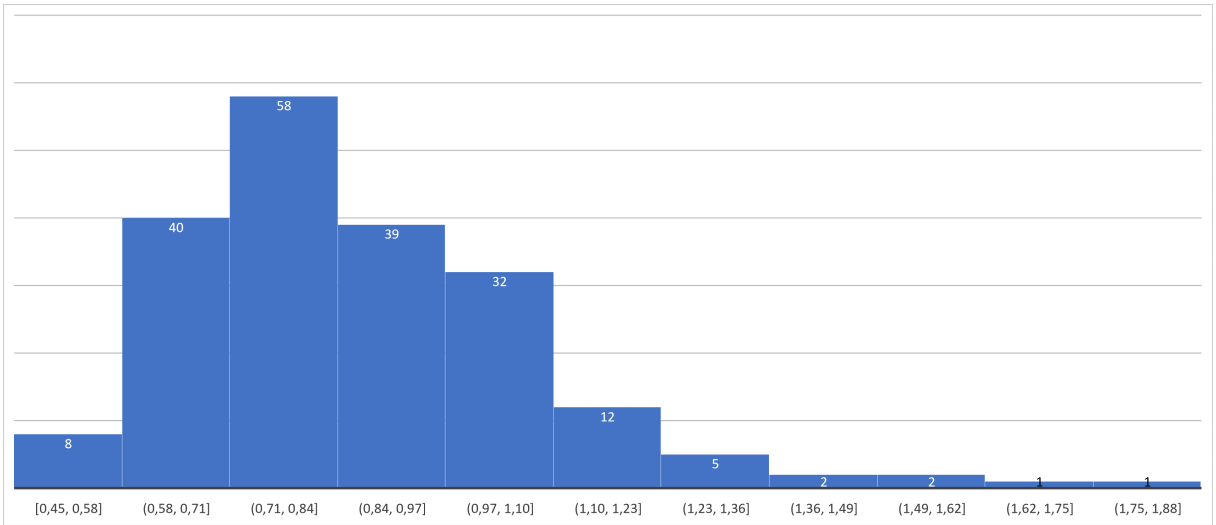


Figure 11: Distribution of validation loss of 200 trials on weekly resampled data

From this graph, it is evident that the validation loss was not the same for all configurations. The different configurations lead to different validation losses. Optuna tries to optimize the search over time by using the Tree-structured Parzen Estimator algorithm to narrow in on a good set of hyperparameters.
The best configuration may vary depending on how many days we want to predict for. The following shows the best configuration we found for weekly resampled data after running 200 trials multiple times.

```
n_timesteps : 101
batch_size : 64
epochs : 74
n_nodes1 : 79
n_nodes2 : 27
filter1 : 84
filter2 : 36
kernel_size : 8
learning_rate : 4.53714E-05
dropout_probability : 0.161184924
use_batch_normalization : True
patience : 26
weight_regularizer : 2.51318E-09
```

We can use this configuration if we want to make a prediction for one week over weekly resampled data.

6 Results

The goal of the model is to forecast the inflow for a water reservoir. The length of the forecast can be changed using the config file and the setting `n_outputs`. The inflow data provided to us contains data for 18 stations, and multiple forecast times are relevant. Therefore, we decided on yearly forecasts, monthly forecasts (4 weeks to be precise), and weekly forecasts. The data we used for training consists of the downloaded data, pre-processed with our developed data download and data cleaning algorithms, plus the inflow time series as the target value. We performed automatic hyperparameter tuning for 200 trials using the validation set for one station and using the best found hyperparameters for all 18 stations to test the performance of the model for the test set. We measure the quality of the forecast using four metrics, namely Mean Absolute Percentage Error, Root Mean Squared Error, Median Absolute Error and Explained Variance (Appendix: 1-4). The training and hyperparameter tuning was performed using mean squared error as the loss function and Adam as optimizer.
All results are reproducible as we have set seed in the beginning of our scripts, using Tensorflows random seed function `tf.random.set_seed()`.

6.1 Yearly forecasts

6.1.1 With hyperparameter tuning

For the yearly forecasts, we focused on the years 2020 and 2021. We first ran the hyperparameter tuning with Optuna for 2021 to get the best hyperparameter based on one station. Afterwards, we reused these hyperparameters to train all stations for 2020 and 2021. Figure 15 and 16 show the best and worst prediction for 2021. Figure 12(b) depicts the average and the median performance for all stations in 2021. Figure 13 and 14 show the best and worst prediction for 2020 and Figure 12(a) shows the average performance for all stations in 2020. The forecast in 2020 for the station Einunna is quite bad. But as we remember Section 1.3 and Figure 1 the model is not able to cope with this heavy drop in the target data.

Metric	Average	Median
Mean Absolute Error (%)	255.87	65.256
Root Mean Squared Error (%)	465.47	102.3265
Median Absolute Error (%)	84.07	29.96
Explained Variance	-122.59	0.5385

(a) Average and Median performance for 2020 forecasts

Metric	Average	Median
Mean Absolute Error (%)	66.85	62.625
Root Mean Squared Error (%)	100.21	101.9445
Median Absolute Error (%)	41.13	36.765
Explained Variance	0.484	0.453

(b) Average and Median performance for 2021 forecasts

Figure 12: Average and Median performance

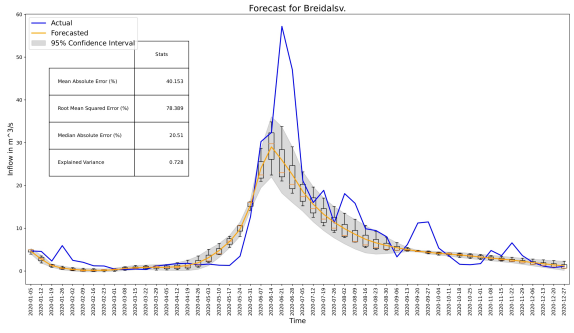


Figure 13: Yearly Forecast for station Breidalsv with hyperparamter tuning

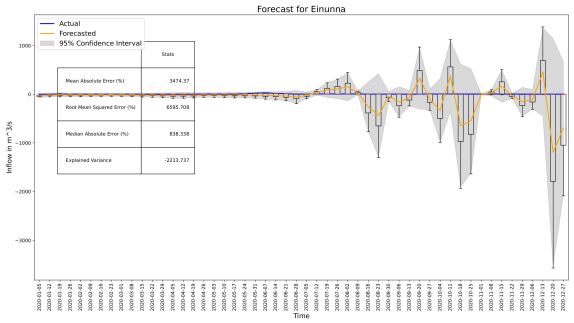


Figure 14: Yearly Forecast for station Einunna with hyperparamter tuning

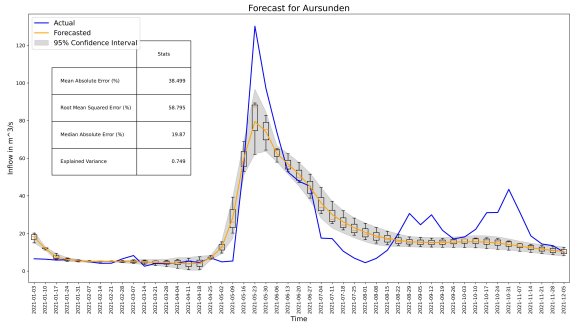


Figure 15: Yearly Forecast for station Aursunden with hyperparamter tuning

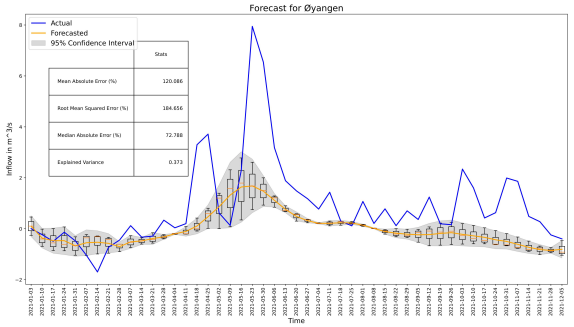


Figure 16: Yearly Forecast for station Øyangen with hyperparamter tuning

6.1.2 Without hyperparameter tuning

In this section, we are running the model without any hyperparameter-tuning to predict the inflow of all stations in the years 2020 and 2021. Here we also did not yet combat overfitting of the model. In table 3 we can see the averages and the medians of the metrics over all stations. The model tries to predict 52 weeks based on the previous 40 weeks. The learning rate is defined as 0.001, not optimized by Optuna. For the year 2020 the station Einunna is an outlier, for which the model's accuracy is quite bad. In addition to the drop of the inflow of the station Einunna as explained in the previous Section 6.1.1, one further reason is that the learning rate is still too high and this might cause an update of the weights which are too drastically. Compared to 14 e.g. the Median Absolute Error of this station is 100 times higher, explaining the huge differences in the averages in table 3 and table 12(a). That is why we also added the medians over all stations.

Metric	Average	Median
Mean Absolute Error (%)	20378.79	64.11
Root Mean Squared Error (%)	37006.05	119.4325
Median Absolute Error (%)	8241.53	35.81
Explained Variance	-1339387	0.297

Table 3: Average and Median performance for yearly forecasts without hyperparameter tuning for the year 2020

The lake with the lowest mean squared error is shown in Figure 17. To get some direct comparison to the predictions with hyperparameter tuning from Section 6.1.1 figure 18 shows the predictions for Breidalsv without any tuning of hyperparameters. We can follow that hyperparameter tuning is indeed leading to better results for the lake Breidalsv 13. The overall performance based on the listed metrics also increased with tuned hyperparameters.

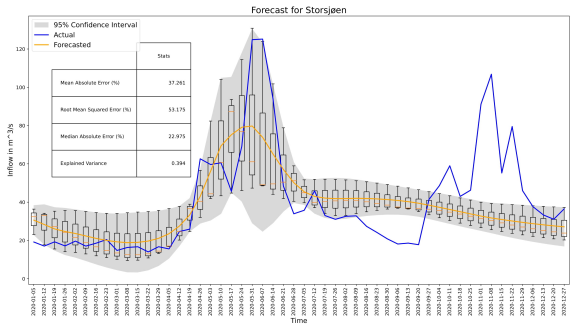


Figure 17: Yearly Forecast for station Storsjøen without hyperparamter tuning

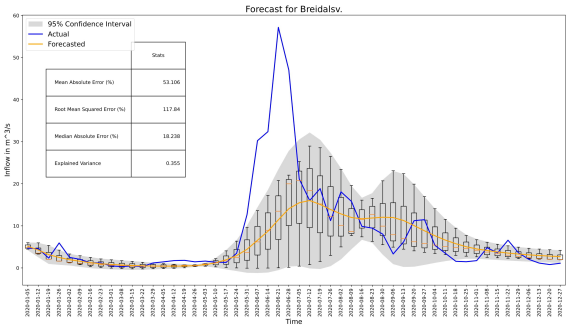


Figure 18: Yearly Forecast for station Breidalsv without hyperparamter tuning

Forecasting 2021 we received following results shown in table 4.

Metric	Average	Median
Mean Absolute Error (%)	72.32	73.95
Root Mean Squared Error (%)	112.17	112.15
Median Absolute Error (%)	41.67	47.03
Explained Variance	0.35	0.30

Table 4: Average and Median performance for yearly forecasts without hyperparameter tuning for the year 2021

Interestingly, we received again the lowest Mean Absolute Error for the lake Storsjøen which is shown in Figure 19. For this year we see that there is just one peak around summer which might indicate some errors in the measurement of the previous year, as normally there is just one peak around spring / summer in each year for the water inflow.
Again, to also get some direct comparison between the predictions of the model with and without hyperparameter tuning, we refer to the results for the lake Aursunden in the year 2021 in Figure 20. We can again infer that hyperparameter tuning has an impact on the accuracy for this station, as the predictions with the given metrics are worse than shown in Figure 15. Overall, we can conclude that with hyperparameter tuning, we get better results for the yearly forecasts.

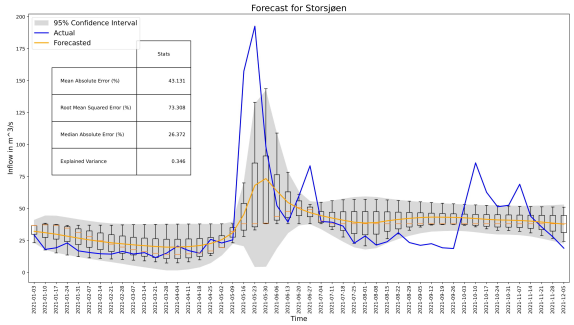


Figure 19: Yearly Forecast for station Storsjøen without hyperparameter tuning

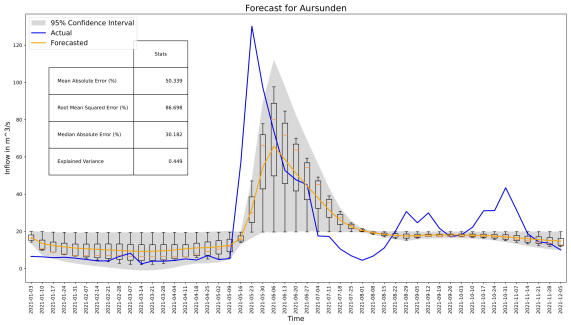


Figure 20: Yearly Forecast for station Aursunden without hyperparameter tuning

6.2 Monthly forecasts

Monthly forecasts are also relevant in the context of inflow prediction. As we are taking weekly averages for the data, we are performing 4-week forecasts. Using the best hyperparameters found by the automatic hyperparameter tuning with Optuna, we tested the performance with all 18 stations. The average performance for all stations is shown in Table 5. The lowest relative mean square error was achieved for the station Fundin, see Figure 21, and the highest one occurred for Øyangen, see Figure 22.

Metric	Average
Mean Absolute Error (%)	75.411
Root Mean Squared Error (%)	84.796
Median Absolute Error (%)	70.077
Explained Variance	-1.375

Table 5: Average performance for monthly forecasts

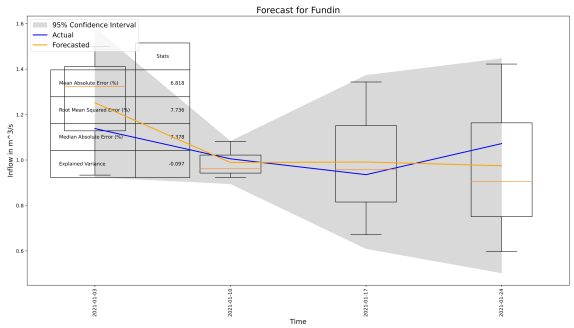


Figure 21: Monthly Forecast for station Fundin

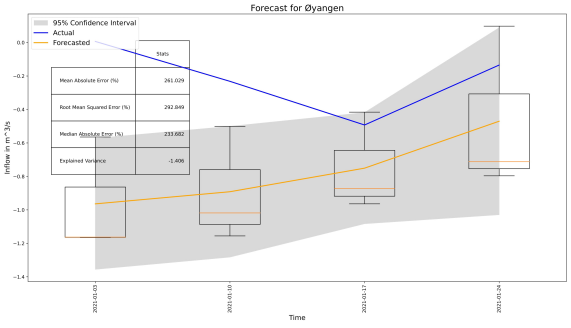


Figure 22: Monthly Forecast for station Øyangen

6.3 Weekly forecasts

The smallest duration we have tested is weekly forecasting. We used both daily and weekly resampled data for this test on the same set of hyperparameters. We found this set of hyperparameters by performing automatic hyperparameter tuning.
Like yearly and monthly forecasting, weekly forecasting also shows different performances for different stations. We calculated Mean Absolute Error, Root Mean Squared Error, Median

Absolute Error, and Explained Variance for all 18 stations provided by the customer and then calculated the average of all those stations.
On daily resampled data, the lowest relative mean square error was achieved for the station Marsjø, see Figure 23, and the highest one occurred for Tesse, see Figure 24.

Metric	Average
Mean Absolute Error (%)	199.2460556
Root Mean Squared Error (%)	236.1223333
Median Absolute Error (%)	209.0207222
Explained Variance	-1.153611111

Table 6: Average performance for weekly forecasts on daily resampled data

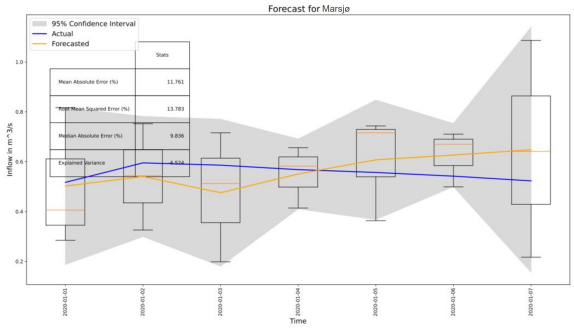


Figure 23: Weekly forecast for the station Marsjø

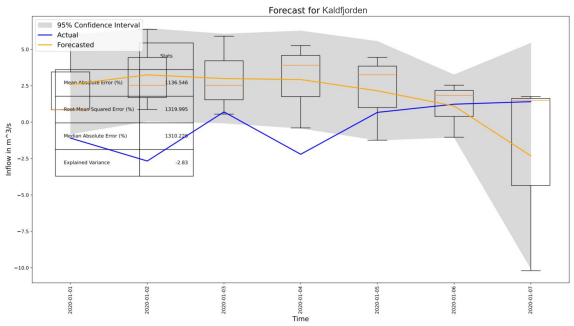


Figure 24: Weekly forecast for the station Kaldfjorden

6.4 Other resampling methods

Instead of considering weekly means, we are now considering coarser methods, like bi-weekly and three-weekly averages. We use the same hyperparameters as in Section 6.1.1 for the forecasts. For the year 2020 we will have a look at „Einunna“ as in Figure 14, which had the worst outcome and accuracy for that specific year. From Figure 26 we can now follow that the data has less noise, so the model is now able to make more accurate predictions on the data and probably weights the drop from Figure 1 less than before. Therefore, the weekly averages of the input columns are calculated and the three-weekly averages of the inflow / target column, resulting in different time resolutions of input and target columns as shown in Figure 27. To fill the gaps, we interpolated the missing values. This is done, to not lose too much input data, overall it results in a much better performance for this particular station. As three weekly means still represent the annual trends of the inflow, it might make sense to consider this option for stations which had a lot of noise in previous years, such as Einunna. The data provider also stated that „Weekly mean values or coarser time resolution is recommended“. The coarser the resampling method, while still being reasonable, the more noise can be removed and the better the model performs. Of course, one has to keep in mind that three weekly averages might not reflect the inflow values as good as weekly averages.

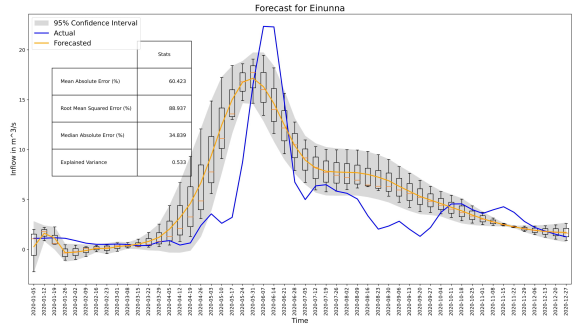


Figure 25: Yearly Forecast for station Einunna (resampled data, bi-weekly)

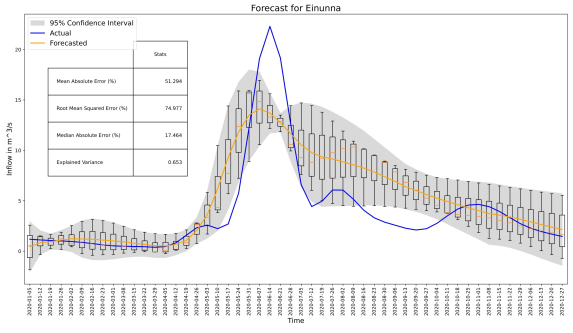


Figure 26: Yearly Forecast for station Einunna (resampled data, three-weekly)

	ute/ swe	Anestølen/ swe	Bakko/ swe	Groset/ swe	Brunkollen klima/ swe	Einunna
2001-12-02T00:00:00.000000000		0.11779	0.10042	0.04628	0.11779	nan
2001-12-09T00:00:00.000000000		0.12417	0.11744	0.05425	0.12417	1.51293
2001-12-16T00:00:00.000000000		0.12723	0.12003	0.04829	0.12723	nan
2001-12-23T00:00:00.000000000		0.13003	0.11948	0.04613	0.13003	nan
2001-12-30T00:00:00.000000000		0.14458	0.12429	0.04916	0.13891	1.60068
2002-01-06T00:00:00.000000000		0.12708	0.13252	0.04464	0.12571	nan
2002-01-13T00:00:00.000000000		0.14086	0.12932	0.04524	0.14086	nan
2002-01-20T00:00:00.000000000		0.15107	0.14749	0.05103	0.15107	0.91878

Figure 27: Different Resampling methods on input (1W avg) and target (3W avg, column Einunna) data

The same holds for the year 2021 and station Øyangen. The accuracy increases slightly when resampling the data to three weekly averages as depicted in Figure 28 compared to Figure 16.

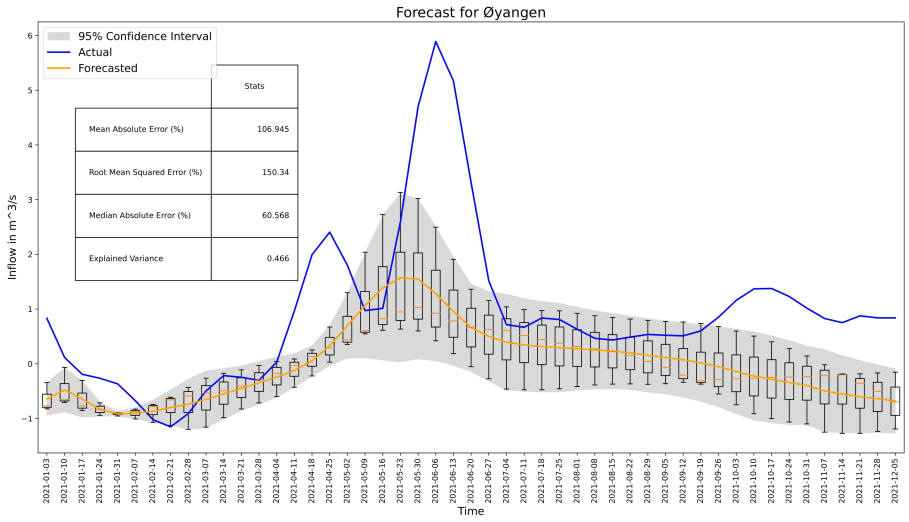


Figure 28: Yearly Forecast for station Øyangen (resampled data, three-weekly)

6.5 Sum over all stations and second data source

One further interesting view is the sum over all stations, that is of special interest for managers who control the water resources over a whole region. Summing up all columns of the inflow data, meaning building the sum of the inflows of all reservoirs, results in removing some noise from the data. The performance for yearly forecasts is depicted in Figure 29 and Figure 30.

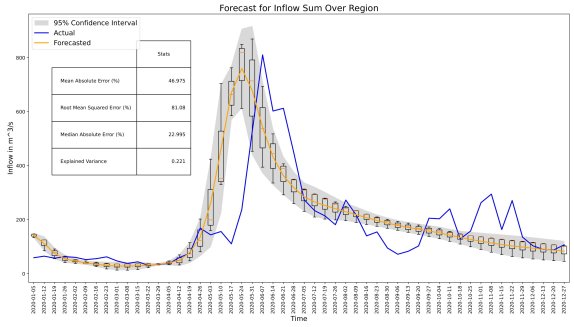


Figure 29: Yearly Forecast for the sum over all stations, 2020

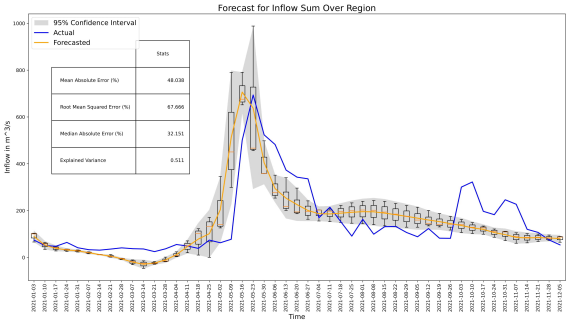


Figure 30: Yearly Forecast for the sum over all stations, 2021

As mentioned in Section 2.1 we also received one further water inflow time series of the lake Sula. The time series for this station contains overall not so much noise, resulting in a good accuracy. Using the tuned hyperparameters from Section 6.1.1 we received the following yearly predictions.

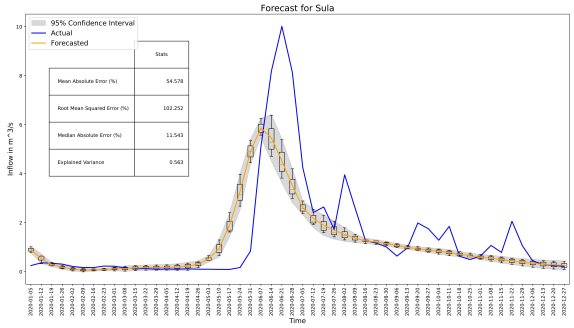


Figure 31: Yearly Forecast for station Sula, 2020

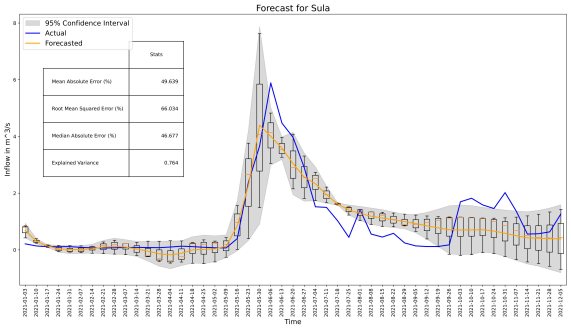


Figure 32: Yearly Forecast for station Sula, 2021

7 Project timeline

The project kick-off meeting was on the 15th of November. After receiving the project requirements and other relevant documents, we started analyzing those. We divided our project development into six main parts.

- Knowledge gathering and project setup: At first, we started reading papers and documents relevant to the project. We spent approximately one week on this task. This stage was also used to understand the code by Herbert et al. [6] and find its shortcomings.
- Developing `DataLoader`: In this module, we created the `DataLoader` class and many functionalities to download data for different stations. We spent approximately four weeks to finish this module and incrementally added more attributes and additional features like data interpolation.
- Model development: This is the most vital project part. We created the TensorFlow model, restructured the code, and added plots. We also improved the model throughout the project duration by adding techniques to combat overfitting and making the model more flexible with the versatile config file. As changes to the model architecture to combat overfitting and improve the performance were made throughout the whole project, work on this part continued throughout the whole project time.
- Testing: We started testing and analyzing the model as soon as the basic model was ready. We continued this process throughout the remaining project time.
- Automatic hyperparameter tuning: During model development and testing, we noticed that the hyperparameters have an important effect on the model performance. As the manual process was deemed as time-consuming and tedious, we used Optuna to tune the hyperparameters automatically. Adding this framework, finding ways to combat overfitting and running it for the different prediction windows took about 3 weeks.
- Documentation: On each block of the code, we added docstrings to improve the readability. We have also written a formal report, and we started writing it at the beginning of the semester and finished it in 12 weeks.

8 Conclusions

The model developed for this project to predict water inflow in a reservoir is based on the code of Herbert et al. [6]. We adopted the model architecture trained on U.S. data from the open-source code published by the authors and applied it to corresponding Norwegian data. In general, we obtained reasonable results at the end of the project. However, due to noise in the Norwegian data and other difficulties described below, it was more difficult to make accurate predictions than with the smooth U.S. inflow data.

To be able to train the model with Norwegian data, we developed a `DataLoader` to download data from a Norwegian open source website for specific configurations defined in the configuration file. Thereby, we noticed that many values were missing in the downloaded data, so we established several techniques to fill in the missing ones. The target data of inflow to a reservoir, i.e., the values we want to predict, were provided by ThinkOutside, the company cooperating with us for this project. Unfortunately, the daily inflow values showed large fluctuations. Therefore, the preferred choice in terms of temporal resolution is weekly averages, but if the fluctuations are still too high for one station, the temporal resolution can be changed to two or three weeks.

We had to clean up and restructure the existing code, as the work by Herbert et al.[6] used some obsolete libraries and a code and repository structure that was not well organized. We split the code into several Python files to improve code quality, improving the structure of the layers and the ML model. To reduce redundant code, we defined a subclass of the TensorFlow layers API that creates the default block of residual CNN layers. We outsourced the configurations required to execute the code by creating config files, as shown in appendix B.1 to quickly debug the code and test which hyperparameters best fit our model.

To monitor the training and validation loss, we have also added the use of Tensorboard. Since we have a very flexible codebase in the sense that we can predict long or short time periods, add different weather features as input to the model, or use different NA filling methods, it was essential to find a well-suited set of hyperparameters for each setting. To this end, we included the search for the best hyperparameters with Optuna. In the automatic hyperparameter search, we found that the best configurations can vary depending on the number of days we want to predict.

Since the goal is to forecast the inflow for a water reservoir, we tested different lengths for the prediction period, which can be easily changed in the config file. In general, overall performance increased with tuned hyperparameters for yearly forecasts, monthly forecasts (4 weeks), and weekly forecasts. However, performance significantly deviated across stations due to variations in data quality for specific stations. In addition, the station with the best accuracy for each time frame changed. To combat the poor performance of stations with a lot of noise in the year prior to the forecast, it is recommended to use coarser methods than weekly means for resampling the data. For regional forecasts, where the sum overall stations is used for the forecast, some noise could be further removed.

In conclusion, depending on the time frame, station and settings, different issues must be considered and solved in order to obtain the best performance of the model, as we deal with data measurements whose quality is not perfect.

We achieved our goal of implementing an easily adjustable code-base to predict water inflow, and the configuration files as well as the automatic hyperparameter tuning enable the user to choose the required settings.

Bibliography

References

[1] Mohamed Abdel-Nasser and Karar Mahmoud. "Accurate photovoltaic power forecasting models using deep LSTM-RNN". In: *Neural Computing and Applications* 31.7 (2019), pp. 2727–2740. ISSN: 1433-3058. DOI: 10.1007/s00521-017-3225-z.

[2] Faraz Malik Awan, Roberto Minerva, and Noel Crespi. "Improving Road Traffic Forecasting Using Air Pollution and Atmospheric Data: Experiments Based on LSTM Recurrent Neural Networks". In: *Sensors* 20.13 (2020). ISSN: 1424-8220. DOI: 10.3390/s20133749. URL: https://www.mdpi.com/1424-8220/20/13/3749.

[3] Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting*. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-29852-8. DOI: 10.1007/978-3-319-29854-2.

[4] TensorFlow/François Chollet. *The Functional API*. https://www.tensorflow.org/guide/keras/functional.

[5] Noel B. Elizaga, Elmer A. Maravillas, and Bobby D. Gerardo. "Regression-based inflow forecasting model using exponential smoothing time series and backpropagation methods for Angat Dam". In: *2014 International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management (HNICEM)*. IEEE, 112014, pp. 1–6. ISBN: 978-1-4799-4020-2. DOI: 10.1109/HNICEM.2014.7016185.

[6] Zachary C. Herbert, Zeeshan Asghar, and Carlos A. Oroza. "Long-term Reservoir Inflow Forecasts: Enhanced Water Supply and Inflow Volume Accuracy Using Deep Learning". In: *Journal of Hydrology* 601 (2021), p. 126676. ISSN: 0022-1694. DOI: https://doi.org/10.1016/j.jhydrol.2021.126676. URL: https://www.sciencedirect.com/science/article/pii/S0022169421007241.

[7] International Energy Agency. *Hydropower Special Market Report*. 2021. DOI: 10.1787/07a7bac8-en. URL: https://www.oecd-ilibrary.org/content/publication/07a7bac8-en.

[8] International Energy Agency. *World Energy Outlook 2021*. 2021. DOI: 10.1787/14fcb638-en. URL: https://www.oecd-ilibrary.org/content/publication/14fcb638-en.

[9] Kartverket, Geovekst, Kommuner, Corine og OSM - Geodata AS — NVE. URL: https://sildre.nve.no/map?maxAge=24&basemap=LANDCSCAPE&x=872254&y=6628413&zoom=2.

[10] Donald R. Kendall and John A. Dracup. "A comparison of index-sequential and AR(1) generated hydrologic sequences". In: *Journal of Hydrology* 122.1 (1991), pp. 335–352. ISSN: 0022-1694. DOI: https://doi.org/10.1016/0022-1694(91)90187-M. URL: https://www.sciencedirect.com/science/article/pii/002216949190187M.

[11] Taewook Kim and Ha Young Kim. "Forecasting stock prices with a feature fusion LSTM-CNN model using different representations of the same data". In: *PloS one* 14.2 (2019), e0212320. DOI: 10.1371/journal.pone.0212320.

[12] Frederik Kratzert et al. "Rainfall–runoff modelling using Long Short-Term Memory (LSTM) networks". In: *Hydrology and Earth System Sciences* 22.11 (2018), pp. 6005–6022. DOI: 10.5194/hess-22-6005-2018.

[13] Shiu Kumar et al. "Forecasting the spread of COVID-19 using LSTM network". In: *BMC bioinformatics* 22.6 (2021), p. 316. DOI: 10.1186/s12859-021-04224-2.

[14] Le et al. “Application of Long Short-Term Memory (LSTM) Neural Network for Flood Forecasting”. In: *Water* 11.7 (2019), p. 1387. DOI: 10.3390/w11071387.

[15] Jun Liu et al. “TD-LSTM: Temporal Dependence-Based LSTM Networks for Marine Temperature Prediction”. In: *Sensors* 18.11 (2018). ISSN: 1424-8220. DOI: 10.3390/s18113797. URL: https://www.mdpi.com/1424-8220/18/11/3797.

[16] Alireza Yekta Meyal et al. “Automated Cloud Based Long Short-Term Memory Neural Network Based SWE Prediction”. In: *Frontiers in Water* 2 (2020), p. 53. ISSN: 2624-9375. DOI: 10.3389/frwa.2020.574917. URL: https://www.frontiersin.org/article/10.3389/frwa.2020.574917.

[17] Dimitris M. Papamichail and Pantazis E. Georgiou. “SEASONAL ARIMA INFLOW MODELS FOR RESERVOIR SIZING1”. In: *JAWRA Journal of the American Water Resources Association* 37.4 (2001), pp. 877–885. ISSN: 1093474X. DOI: 10.1111/j.1752-1688.2001.tb05519.x.

[18] Ridwan Siddique and Alfonso Mejia. “Ensemble Streamflow Forecasting across the U.S. Mid-Atlantic Region with a Distributed Hydrological Model Forced by GEFS Reforecasts”. In: *Journal of Hydrometeorology* 18.7 (2017), pp. 1905–1928. ISSN: 1525-755X. DOI: 10.1175/JHM-D-16-0243.1.

[19] Timo Stöttner. *Why Data should be Normalized before Training a Neural Network*. https://towardsdatascience.com/why-data-should-be-normalized-before-training-a-neural-network-c626b7f66c7d. Accessed: 2021-12-16.

[20] Qianlong Wang et al. “Earthquake Prediction Based on Spatio-Temporal Data Mining: An LSTM Network Approach”. In: *IEEE Transactions on Emerging Topics in Computing* 8.1 (2020), pp. 148–158. DOI: 10.1109/TETC.2017.2699169.

[21] Dan Welsby et al. “Unextractable fossil fuels in a 1.5 °C world”. In: *Nature* 597.7875 (2021), pp. 230–234. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03821-8.

[22] Wikipedia contributors. *Norwegian Water Resources and Energy Directorate — Wikipedia, The Free Encyclopedia*. [Online; accessed 10-December-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Norwegian_Water_Resources_and_Energy_Directorate&oldid=1058427463.

9 Appendix

A Requirements to run the Code

In order to run the code, you have to make sure to have Anaconda, Python and pip installed on your local computer. As the data we are training our model on is not very big (weekly time-series starting in the year 2000), it is sufficient to run the code on your local computer. Also, due to the small data size, running the Code on the GPU doesn't increase the runtime speed.
As an IDE, we would recommend to use Pycharm due to its easy handling and debugging options. Not to run into any package-version issues, we have created a `requirements.txt` file which contains all python-packages needed to run the code. To install these packages, we suggest creating a virtual environment to run all scripts. For that, open a terminal and run:

```
$ python3 -m venv /user/some_path/
$ /user/some_path/activate.bat
$ cd /user/repository_path/
$ pip install -r requirements.txt
```

Now you should have your own virtual environment with all needed packages installed. As a last step, you have to add the virtual environment as an interpreter to your Pycharm configurations.

B Project structure

The project contains three Python scripts that can be run by the user. After the requirements from `requirements.txt` are installed, the user can change the config files to adjust the settings before executing them. The three executable scripts are:

1. `download_data/download_data.py`
2. `ResCNN_LSTM/ResCNN_LSTM_fit.py`
3. `ResCNN_LSTM/hyperparameter_tuning.py`

The first script is responsible for downloading the data from `sildre.nve.no` as previously described. The settings for this data download can be found in `download_data/config.yaml` which includes the filename of the files created by the script. The script will print the name of all used stations and attributes read from the config, so the user is aware of what is being downloaded. In addition to that, the user can also set the option to run further preparation steps. According to the defined configurations, different methods are executed to fill missing values. Afterwards, plots are created to display how well the methods filled the missing values.
The second script is used to train the model with one specific configuration which can be defined by the user in `ResCNN_LSTM/config.yaml` which includes the file paths for the input files (training and testing data) as well as the location of the output files (predictions and plots). During runtime, the progress of the training is printed and also logged to Tensorboard.
The third script is used for the previously described automatic hyperparameter tuning with Optuna. It uses `ResCNN_LSTM/config.yaml` as a config and tries out multiple different adjusted configs. All these generated configs are also saved in the folder `ResCNN_LSTM/hypertune_configs` once they are generated and used. The results of the hyperparameter tuning are printed but also written to the file `ResCNN_LSTM/hypertune_configs/configs_evaluation.csv`.

B.1 Configuration files

The configuration files, often abbreviated as config files, define the parameters, options, and initial settings for running the ML model. The files are formatted to allow the user to configure and modify them.

download_data/config.yaml

```yaml
station_name:
# A list of available stations to download weather data
  - Øvre Leirbotn
  - ØVERBYGD II
  - Siccajavvre
  - Storstilla ndf
  - Namsvatn tunnel
  - Svarttjønnbekken Klima
  - Maurhaugen - Oppdal
  - Grimsa snøpute
  - Anestølen
  - Kyrkjestølane
  - Vetlebotn
  - Bakko
  - Groset
  - Brunkollen klima
  - Nedre Sjodalsvatn
  - Øvre Heimdalsvatn

features:
# Weather features we can download
  - swe
  - temperatur
  - relative humidity
  - wind direction and speed
  - precipitation
  - wind speed
  - snow depth
  - ground water level
  - discharge
  - reservoir volume

start_time:
  - <dd>
  - <mm>
  - <yyyy>

end_time:
  - <dd>
  - <mm>
  - <yyyy>

output_path:
# Specify in which folder to store the data
  "../data"

save_name:
  "data_swe_weather.csv"

time_resolution:
  - day
  # can also be set to [hour, as_measured]


# Configurations for preparation steps
run_preparation: <True/False>
    # Run further preparation steps of downloaded data (fill missing values)
drop_cols: 0.55
    # <float>, Drop all columns which contain more NA values than given share
threshold_linear_inter: 0.55
    # <float> If Total Na Values smaller than threshold and consecutive_days_with_nulls
    smaller than defined, then run interpolation, otherwise run fill_nan_method
consecutive_days_with_nulls: 90
```

```
    # <int> If the max of consecutive days with Null values for given station/ year is
    less than defined, then interpolate
fill_nan_method: "fill_nan_by_same_features_avg"
    # ["fill_nan_by_daily_avg","fill_nan_by_same_features_avg"], refer to Chapter 2


# Specify how to interpolate
interpolation_method: "linear"
    # How to interpolate NA Values
order_interpolation: <int>
    # Which order to use for interpolation (if interpolation in ["polynomial", "spline"],
    can still be defined, even though e.g. linear interpolation doesn't require order)

# Specify as list of strings which columns to plot before/ after interpolation
plot_columns: ["<station_name>/ <feature>"]
```

ResCNN_LSTM/config.yaml

```
# General specifications
run_on_gpu: <False/True>
    If set to False, pls make sure that you open a new Python Console

#data:
filepath_input: "../data/data_swe_weather.csv"
    # Specify the path to Downloaded SWE/ Temperature Data
filepath_inflow: "../data/inflow-glomma.xlsx"
    # The path of target inflow data
skiprows: 1
    # integer, how many rows to skip in inflow data (for sula:0, inflow-glomma: 1)
index_col: 0
    # Which column to use as index for the pd.DataFrame. Always choose Date column
keep_other_inflow_columns: <False/True>
    # If set to false, all other columns from inflow data are dropped
target_name: ["Storsjøen"]
    # Inflow of which reservoir lake we want to predict, if multiple columns are defined in list,
    these columns are added up, shortcut for all columns is: ["all"]. Interesting for manager of
    whole region
resample_method_inflow: "1W"
    # Take the average inflow of the target within given time resolution (reference to pandas
    resample function for possible values)
resample_method_input: "1W"
    # How to resample the input (SWE, temperature, Percipitation...) Data, as data is read it
    is on daily resolution
interpolate_target: "polynomial"
    # how to interpolate NA values if input and target have different resampling methods
order_interpolation: <int>
    # which order we want to use for interpolation (if interpolate_target is set to
    "polynomial"/ "spline"

# Time Settings
split_y_test: <True/False>
    # If set to true, we split the data into training (=actual training and validation set)
    and test set, if set to False all available data is used as training
    (e.g. to make prediction in future): Take all datapoints until the last 'n_timesteps'
    as training and the last 'n_timesteps' as input to make the prediction
    # NOTE: To make actual future prediction: Set split_y_test to false, as we don't want to
    have a test set, Then training_input_end_date is ignored
training_input_end_date: <yyyy>-<mm>-<dd>
    # We are making a prediction for "n_outputs" upcoming weeks
    (all datapoints <= are used to train the model)
n_timesteps: <int>
    # n_timestamps length of input window, 40 weeks (in case resampling method=1W)
n_outputs: <int>
    # n_outputs how many weeks we want to predict (in case resampling method=1W)


# model settings:
repeats: <int>
    # how often we want to repeat the training and fitting of model
epochs: <int>
n_nodes1: <int>
```

```
    # the dimensionality of the output space
    (i.e. the number of output filters in the convolution) for the first layers block
n_nodes2: <int>
    # the dimensionality of the output space
    (i.e. the number of output filters in the convolution) for the second layers block
filter1: <int>
    # the dimensionality of the output space
    (i.e. the number of output filters in the convolution) for the third layers block
filter2: <int>
    # the dimensionality of the output space of first Dense Layer
kernel_size: <int>
    # specifies the length of the 1D convolution window
learning_rate: <float>
    # the learning rate to control how much to change the model in response to the estimated
    error each time the model weights are updated
dropout_probability: <float>
    # the probability to drop values for the dropout layers; value of 0 is equivalent to no
    dropout
use_batch_normalization: <False/True>
    # apply batch normalizations for the convolutions
batch_size: <int>
monitor: <"val_loss"/"loss">
    # "val_loss" or "loss": what to monitor for early stopping: Stop training when the
    monitored metric has stopped improving
patience: <int>
    # Number of epochs with no improvement after which training will be stopped.
loss: "mse"
    # Loss function. string (name of loss function)
optimizer: "adam"
    # String (name of optimizer)

#L2 regularization/weight decay
weight_regularizer: <float>
    # every coefficient in the weight matrix of the layer will add
    weight_regularizer * weight_coefficient_value**2 to the total loss of the network

#plot and predictions:
run_plot: <True/False>
    # run make_plots, yes or no
path_fc: "../plots/forecast_plot.pdf"
    # save forecast plot as pdf in this file
path_lc: "../plots/loss_curve_plot.pdf"
    # save loss curve plot as pdf in this file

# layer_using
rescnn: [True, True, True]
    # For 3 ResCNN layer-blocks, True means using this layer, False means not using this layer
lstm: [True, True, True, True]
    # How many LSTM-Layers to use
path_predictions: "../data/prediction.csv"
    # Where to store the predictions of Model
```

B.2 Tensorboard

To visualize the (validation)-loss curve using Tensorboard you have to run the following commands in your Terminal. For every repeated run, there is one folder named with the current unix-timestemp created within the `ResCNN_LSTM/logs` directory.

```
$ cd /user/repository_path/ResCNN_LSTM
$ tensorboard --logdir=logs/fill_unix_timestemp
```

Now in the console the user will see that Tensorboard is hosted on the localhost with the portnumber 6006 „http://localhost:6006/“. Navigating to the link will open the Tensorboard surface.

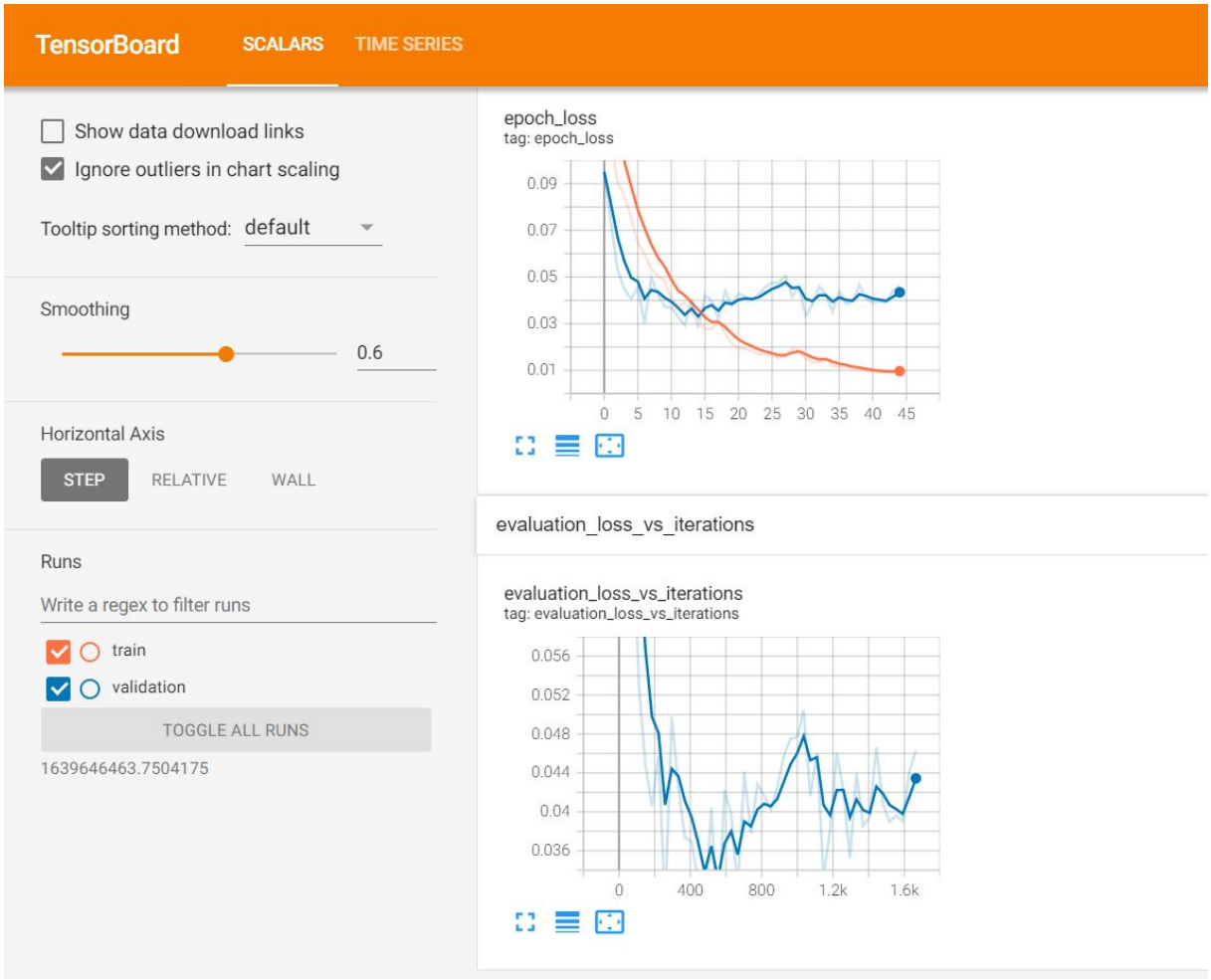


Figure 33: Usage of Tensorboard to track the losses

C Python helper functions

C.1 Data loader

Within the Data Loader class, we have defined several functions. There are some controlling functions, such as `is_valid_start_date` which checks if the date specified in the `config.yaml` is a valid start date, as well as some helper functions which create the download links for every station. Once that is done, the data is downloaded and stored as a CSV file.
If the user wants to further prepare the downloaded data, in the sense of filling missing values, the parameter `run_preparation` needs to be set to `True`. Depending on the selected methods and configurations as explained in B.1, different functions are executed to fill the missing values.

C.2 Model fitting

Within the script `ResCNN_LSTM/ResCNN_LSTM_fit.py` we have also defined some helper functions.

1. `prepare_data_norwegian`
2. `sliding_window_input`
3. `sliding_window_output`

4. `get_stats`

5. `make_plots`

The first function `prepare_data_norwegian` will merge the input and the output data. It also sorts the dataframe by date in order for the upcoming functions to correctly create the input-/output-windows. Afterwards, the function will calculate the averages over the specified resampling-method, e.g. „1W“ for weekly averages. Next, all columns are scaled to values between 0 and 1 using the `MinMaxScaler` from the `sklearn.preprocessing` API. If `split_y_test` is set to `True`, indicating that we will have a training and a test set for the input and the output data, the function will also split the data according to the specifications of `training_input_end_date`, `n_timesteps` and `n_outputs` in the config file. The function returns the training-set, the input of the test set, and the output / target of the test set, as long as `split_y_test` is set to `True`. Otherwise, it would return just a list of the target dates of our prediction instead of the target-test set.
`sliding_window_input` creates the window inputs, as already described in Section 1.3. Iterating over the range of $nrows(training\_set) - config["n\_timesteps"]$ will then return the input vector. One iteration step returns one window. Let us assume we want to predict 20 weeks based on the last 15 weeks. For the training-dataframe of all input variables of shape $(956, 52)$, this would result in an array of shape $(941, 20, 52)$. This means we have $941 = 956 - 20 - 15$ input-windows of length 20.
Following the same principle in `sliding_window_output`, we then receive the target array of shape $(941, 15)$. That means we also have 941 output windows of length 15 as we want to predict 15 weeks, based on the last 20 weeks.
The function `get_stats` will calculate the following metrics, where Y_i are the actual values and $\hat{Y}_i$ indicate the estimated values.

$$MAEn = \frac{\frac{1}{N}\sum_{i=1}^{N}|Y_i - \hat{Y}_i|}{\frac{1}{N}\sum_{i=1}^{N} Y_i} \tag{1}$$

$$RMSE = \frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2}}{\frac{1}{N}\sum_{i=1}^{N} Y_i} \tag{2}$$

$$MedianAbsoluteError = \frac{median(|Y_1 - \hat{Y}_1|, ..., |Y_N - \hat{Y}_N|)}{\frac{1}{N}\sum_{i=1}^{N} Y_i} \tag{3}$$

$$ExplainedVariance = 1 - \frac{\sum_{i=1}^{N}|Y_i - \hat{Y}_i|}{\sum_{i=1}^{N}(Y_i - (\frac{1}{N}\sum_{j=1}^{N} Y_i))^2} \tag{4}$$

The first three metrics are normalized, in the sense of dividing by the mean of Y_i.
The last function `make_plots` will create three different plots. One for the model's learning curve, in case the user decides not to use Tensorboard, which will display both validation loss and training loss. The forecast plot will display the forecasted values against the actual values, again as long as `split_y_test` is set to `True`. If `repeats` > 1 the plot will also show a boxplot and a 95%-confidence-interval of the predictions, as each repeat and prediction is independent of the previous run. The forecasted value is then the average over all runs. The fitting is repeated to reduce randomness in the predictions.