Recursive Time Series Prediction Using Expanding Window Cross Validation

Daniël Jochems David Huizinga Niek Grimbergen Daan van Dam

***Abstract*—This paper investigates how to forecast univariate time series using deep learning, focusing on the question: how should models be evaluated when future predictions recursively depend on prior predictions? We implement a recursive evaluation strategy combined with expanding window cross-validation, where models must predict sequences without access to future ground truth. Using this framework, we train a Long Short-Term Memory (LSTM) model, performing grid search hyperparameter tuning over epochs, hidden units, and lag lengths. Our final model, using 35 lags, 50 hidden units and 50 epochs achieves a mean squared error (MSE) of 2766.98 and a mean absolute error (MAE) of 38.49 on held-out test data. We also briefly explore a Temporal Convolutional Neural Network (TCNN).**

I. Introduction

Predicting future values of a sequence based on past observations, is quite a challenge, especially when the data is noisy, non-stationary, or exhibits long-term dependencies. Recurrent neural networks (RNNs) such as Long Short-Term Memory (LSTM) networks, are powerful tools to tackle this problem due to their ability to capture temporal patterns and nonlinear relationships.

For this assignment, we were provided with univariate time series data and were tasked with designing a predictive model that could generate accurate predictions of future values. The key difficulty in this task is not just fitting a model to historical data, but ensuring that predictions remain stable and reliable as they are recursively used to forecast further into the future. To this end, it is crucial to evaluate the model in a manner that reflects this recursive nature.

In this report, we describe the methodology we followed to engineer training data with lag features, apply expanding window cross-validation, and build models using LSTM and Temporal Convolutional Neural Networks (TCNN). We report on the hyperparameter tuning process, recursive prediction strategy, and performance metrics. Our results highlight how recursive validation aligns better with the prediction task and reveal the performance trends across model configurations.

II. Methods

A. Training data engineering

First, we prepare the timeseries data to accommodate lag features (past values of a variable at a previous time step) using a lag-based sliding window approach. We do this by creating a new dataframe with the lagged values of the time series and dropping the rows with missing values.

Because the objective of this model is to predict future values of the time series, we need to make sure that the model is trained on past values and validated on future values. Every fold, the train fold grows by including the entire previous fold's worth of data and adding the next fold's train fold. The validation fold remains the same size. The validation fold is structured in the following manner: For a given time series of length T, we generate input-output pairs (X_t, y_t) such that the input vector X_t contains the previous n observations $[x_{t-n}, x_{t-n+1}, \ldots, x_{t-1}]$ and the next value to be predicted is $\hat{y_k}$, which is the value at time t. X_t is then updated by removing the first value and inserting $\hat{y_k}$ at the end of the vector giving us $[x_{t-n+1}, \ldots, x_{t-1}, \hat{y_k}]$, and we predict the next value $\hat{y_{k+1}}$. This is repeated until the end of the validation fold is reached (see Figure 1). This recursive structure of predicting values into the future by including already predicted values for subsequent predictions reflects the problem of predicting future values.

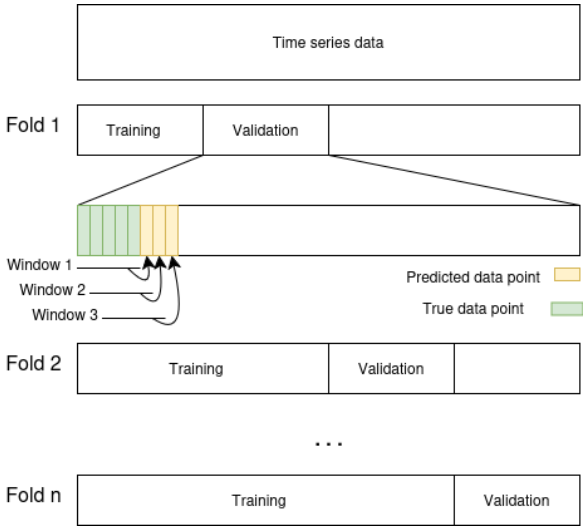


Fig. 1: The recursive structure of the validation fold. The window of data points that are used to predict the next value is shown. The first window has only true data points, subsequent windows increasingly include predicted values until only predicted values are considered. The size of the train fold increases with every fold, while the validation fold remains the same size.

B. Long Short-Term Memory (LSTM)

We used a Long Short-Term Memory (LSTM) neural network to forecast our time series data, building on the

architecture and tuning strategy described by Vien et al. [1]. Since we are working with a univariate time series, the model was set up to predict future values based solely on past observations.

Before training, the library we used standardised the data using z-scores to ensure consistent scaling. We then performed a three-dimensional grid search to tune the model, exploring different combinations of the number of training epochs, the number of hidden units in the LSTM layer, and the number of lagged time steps used as input. This helped us find the setup that worked best for our dataset.

The LSTM architecture itself was straightforward: it included a sequence input layer, one LSTM layer with a tunable number of units, a fully connected layer, and an output layer. To evaluate the model, we used expanding window cross-validation (also known as forward-chaining), which ensured that the model was always tested on future data it hadn't seen during training — an important detail for time series tasks. Moreover, a dropout option of 0.2 was selected.

We used mean absolute error (MAE) and mean squared error (MSE) to measure performance across folds. The final model was chosen based on the average error, weighted by the size of the training data in each fold.

C. LTSM model parameter initialisation

For efficiency purposes, we save time during our initial search for an appropriate number of epochs by fixing the number of hidden units to 15 and lagged time steps to 25, the result of which can be found in table I. These values are the 'averages' of the values we will iterate over during the later, more extensive grid search. After we find the best epoch value, we do another grid search over the number of hidden units and lagged time steps. We use the same method as before, but now we fix the number of epochs to the best value we found in the previous step.

Epoch	MSE	MAE
10	3135.14	44.80
20	2659.23	40.03
50	2542.56	38.15
100	2432.60	38.28
500	2488.35	38.28
1000	2436.53	37.82

TABLE I: Model performance (MSE and MAE) across different training epochs during initial search

D. Temporal Convolutional Neural Network (TCNN)

Additionally, we shortly experimented with a temporal convolutional neural network (TCNN) [2]. TCNNs are models that utilise 1-dimensional convolutional layers to capture temporal dependencies in data. TCNNs make use of dilated convolutions, where a filter is applied over a larger receptive field by skipping over input values. This allows modelling of long-range dependencies without an increase in computational complexity.

To find the best hyperparameters for our model, we leverage the Optuna Python library [3]. Optuna optimises hyperparameters by iteratively generating parameter configurations with a probabilistic model, and observing which configurations minimise the weighted test loss. This search was performed over 2000 iterations.

To accelerate the optimisation process, we only train the model for 10 epochs each iteration. We utilise a mean square error loss function. We use the Adam optimiser for training. In the TCNN architecture, we use the same number of channels for each layer to maintain a consistent feature dimension throughout the network. This design choice simplifies the optimisation process. Table II below illustrates what hyperparameters were found, as well as in what ranges the search was conducted. The results of this model are described in the supplementary section IV as the validation on the test data was not done in line with the assignment.

Hyperparameter	Range	Step Size	Found Value
number of previous steps	[10, 60]	10	60
channels per layer	[2, 40]	2	40
layers	[2, 5]	1	2
kernel size	[3, 9]	2	3
learning rate	[1e-5, 1e-2]	Log Scale	7.49e-3

TABLE II: Hyperparameter search ranges, step sizes, and found values.

III. Results

A. Grid search

Doing another grid search using 50 Epochs, which we found from our earlier search, we group the results by number of steps looking back 'lags' and number of hidden units. In table III we can see that the lowest MSE is at lag = 15.0 and hidden units = 100. The lowest MAE is at lag = 35.0 and hidden units = 50.

For our LSTM model with EPOCHS $= 50$, hidden units = 50 and number of lags = 35 we find the better model achieving a MAE 38.49 of and a MSE of 2766.98 on the test data (recursively predicted as mentioned in Section II-A). A plotting of our predicted values against the true values of the provided test data are shown in Figure

IV. Discussion

As described in our methods section, we used a recursive approach to predict future values of the time series (see Figure 1). We did this for the entire validation folds length and used this "recursive loss" as a metric to evaluate the model. However, it would have been interesting to compare this with a non-recursive approach (like we did for TCNN in Figure 4a), where we would predict the next value using only the true variables instead of the predicted values. That way, you can't accumulate faults made earlier. An additional limitation in our approach is that we did not use more than one layer, whereas using more could be more beneficial, but greatly increases training time.

Lag	Hidden Units	MSE	MAE
5.0	5	5216.51	51.62
	10	4690.37	49.38
	20	4979.76	53.16
	50	15547.00	81.46
	100	104445.60	139.39
15.0	5	4686.33	53.09
	10	2371759.00	477.48
	20	4746.73	52.84
	50	4764.41	57.50
	100	3581.54	49.32
25.0	5	4031.51	50.03
	10	4458.75	49.04
	20	5126.04	49.71
	50	6044.41	56.84
	100	4737.58	49.49
35.0	5	4963.18	51.81
	10	4913.51	48.33
	20	4654.49	46.13
	50	4955.10	45.42
	100	5091.84	46.26
50.0	5	6657.15	63.24
	10	5439.13	50.24
	20	5220.59	47.16
	50	4836.12	46.71
	100	5222.38	47.17

TABLE III: Model performance (MSE and MAE) for varying lag values and number of hidden units

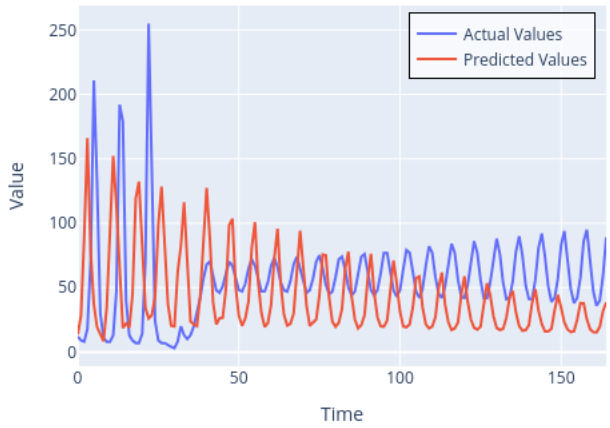


Fig. 2: LTSM predictions vs Actual values. The plotted values are the recursively predicted values by our LTSM model using 50 epochs, 35 lags and 50 hidden units

AI STATEMENT

GitHub Copilot was used to assist in writing this report. It was used to generate code snippets and to help with the writing of the text. Moreover, ChatGPT was used to format the single paper source as a '\bibitem' and the tables in LaTeXformatting.

REFERENCES

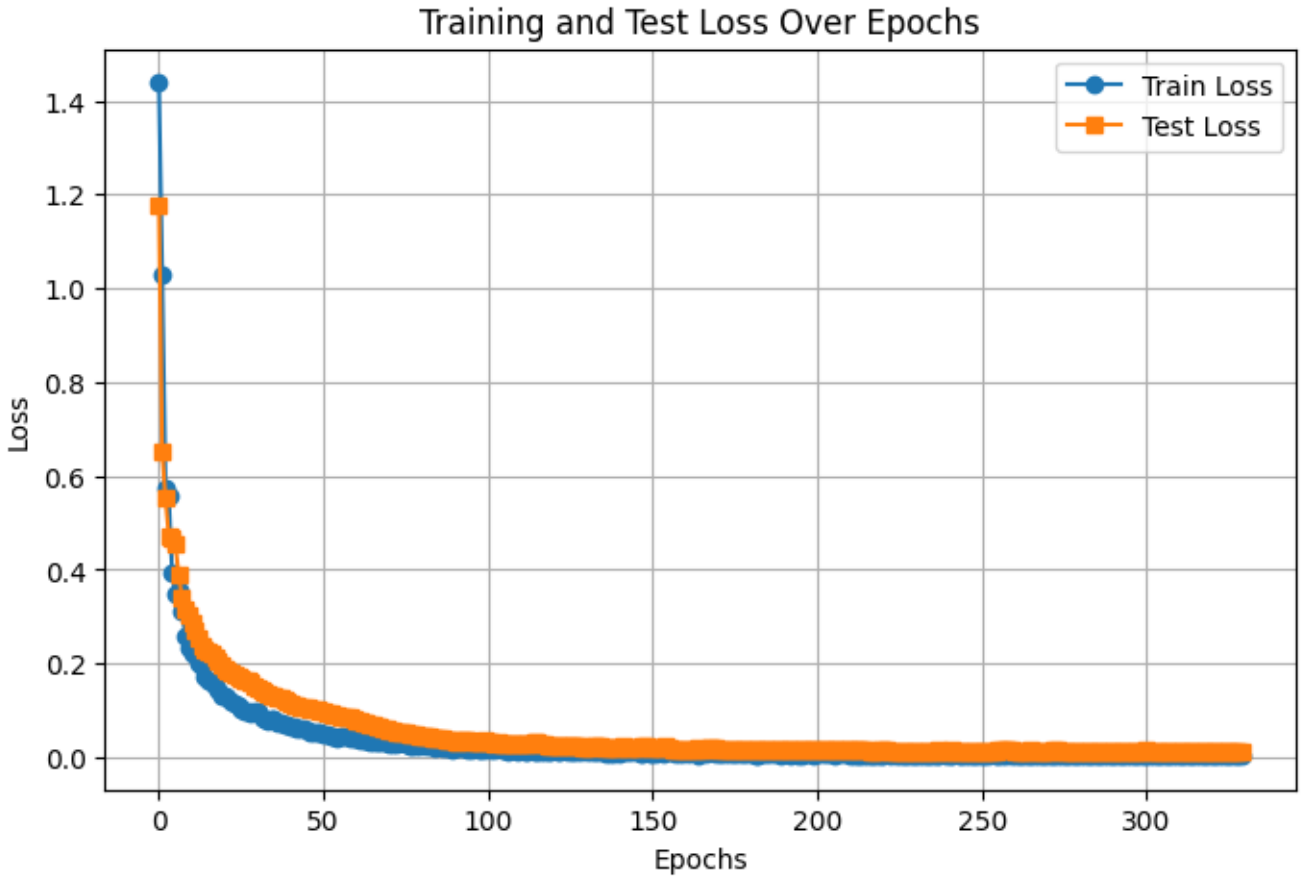
[1] T. Kuen, L. R. F. Rose, W. K. Kong, and B. S. Vien, *A machine learning approach for anaerobic reactor performance prediction using Long Short-Term Memory Recurrent Neural Network*, Materials Research Proceedings, vol. 18, 2021. Publisher: Materials Research Forum LLC.

[2] C. Lea, M. D. Flynn, R. Vidal, A. Reiter, and G. D. Hager, *Temporal convolutional networks for action segmentation and detection*, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 156–165, 2017.

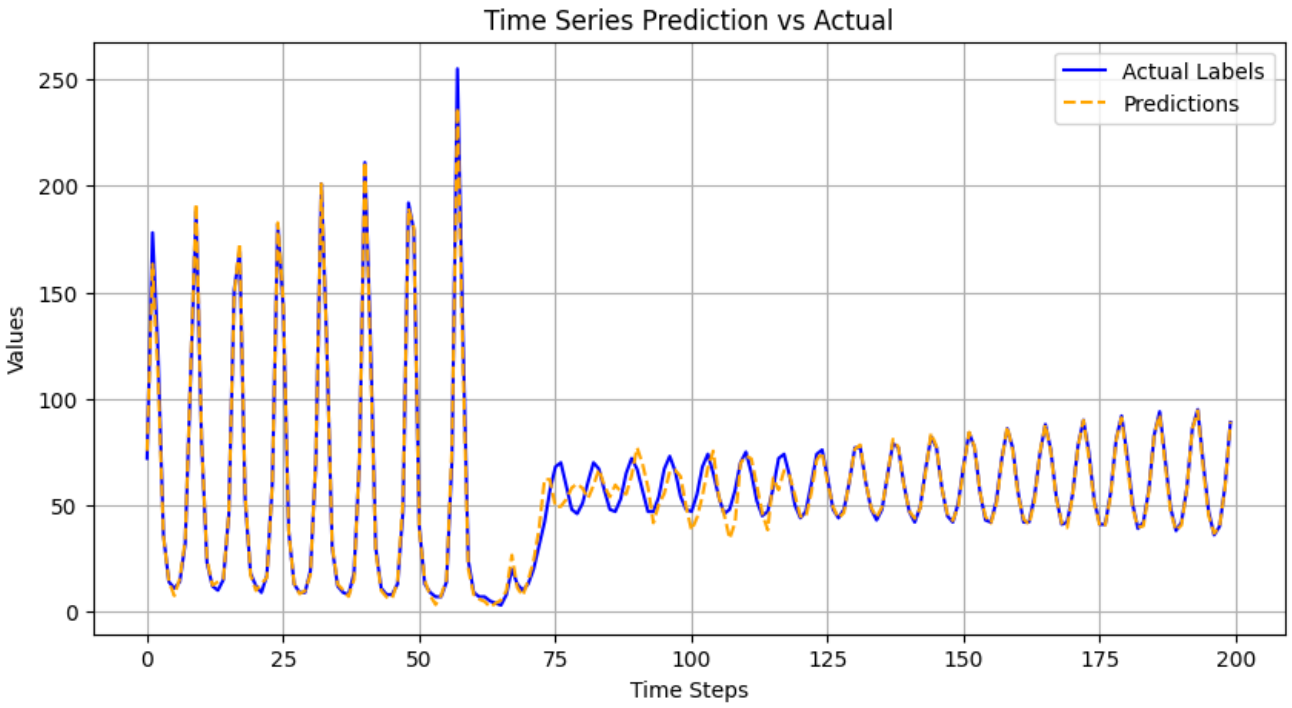
[3] Optuna, *Optuna: A hyperparameter optimization framework*, https://optuna.org, 2019.

SUPPLEMENTARY TCNN RESULTS

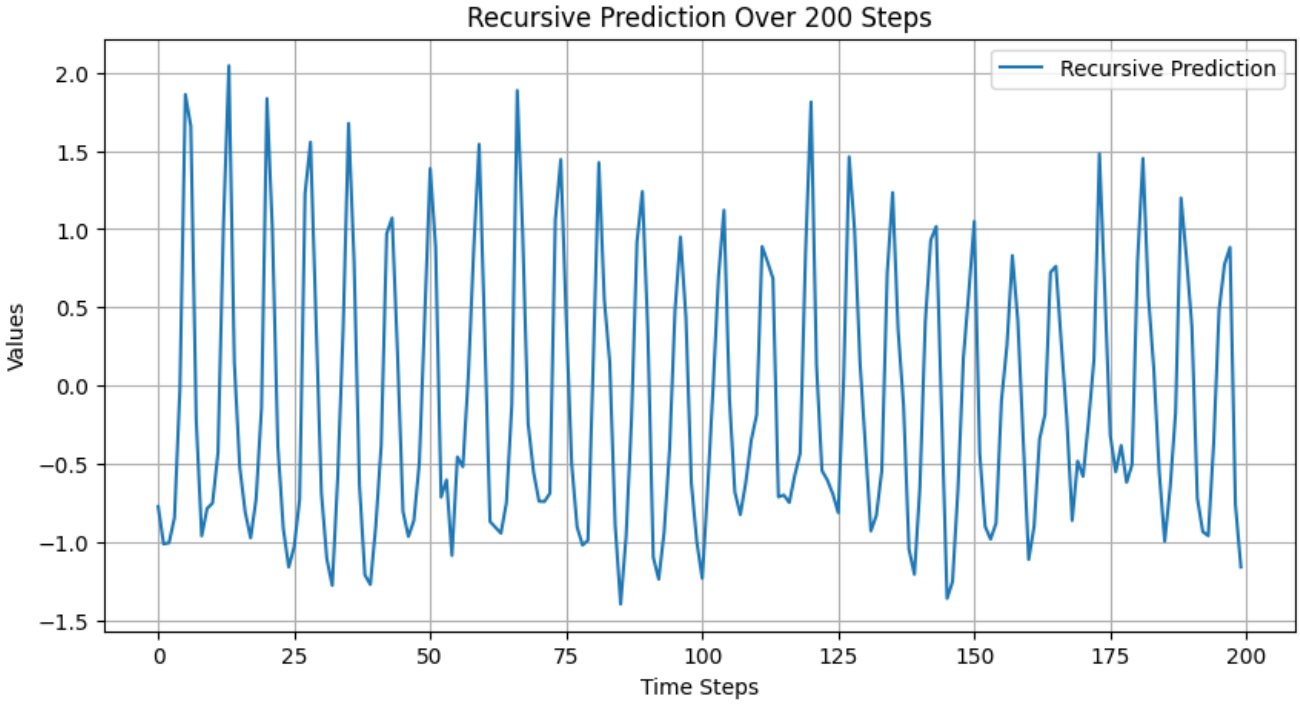
The TCNN model achieved a mean absolute error of 28.68 and a mean squared error of 3.26. Using the found values as described in Table II. It is important to note that in Figure



(a) Train vs. Test Curve. Here are the results produced by the TCNN model of the loss as described in section II-D. It is important to note that these are not recursively predicted values.



(a) Actual vs. Predicted. The non-recursively predicted values plotted against the true test values using the TCNN model.



(b) Recursive Predictions of the TCNN model. These are the next 200 recursively values as produced by the TCNN model.