Technische Universität Berlin

Chair of Database Systems and Information Management

Master's Thesis

Empirical Comparison of Forecasting Methods

Leonardo Araneda Freccero
Degree Program: ICT Innovation
Matriculation Number: 406022

Reviewers
Prof. Dr. Volker Markl
Prof. Dr. Odej Kao

Advisor(s)
Behrouz Derakhshan
Bonaventura Del Monte

Submission Date

27.07.2021

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, 27.07.2021

. . . . . .

Leonardo,Araneda Freccero

Zusammenfassung

Tipps zum Schreiben dieses Abschnitts finden Sie unter [**?**]

Abstract

Time series forecasting is an important tool for businesses, organizations and governments as it enables them to better plan and adapt to the future. Historically, deep learning based forecasting models have been shown to perform poorly compared to classical approaches such as ARIMA, ETS and Theta. In recent years several new deep learning based methods has been proposed and thus evaluating these is important. This thesis investigate several state of the art forecasting models, and methods for how to perform fair, accurate and reproducible comparisons. Since the domain of machine learning research is currently in a reproducibility crisis, particular focus is placed on identifying methods for generating technically, statistically and conceptually reproducible results.

This thesis also introduces Crayon, an open source benchmarking suite for fair, accurate and reproducible comparisons of forecasting methods. Crayon makes use of distributions of error metrics for its benchmarks and ranks algorithms against each other through a novel aggregation method of error distributions. This method, named RMSE4D, is applicable to any error metric distribution and scores algorithms generating consistent forecasts higher. Furthermore, tooling based on the Kolmogorov-Smirnov 2 sample test is employed in Crayon to enable statistical reproducibility of benchmarks and its practical benefits are demonstrated through protecting a forecasting framwork against accuracy regressions. Crayon is used in this thesis to benchmark 13 state of the art forecasting algorithms on four popular public dataset with complimentary characteristics in terms of trend & seasonality. The major findings of this comparison is that DeepAR, Transformer and the N-BEATS Ensamble are the best performing models on three of the datasets but that naive and classical approaches such as Theta are the best performing algorithms for highly trended datasets.

Acknowledgments

For recommendations on writing your Acknowledgments see [**?**].

Contents

List of Figures

List of Tables

List of Abbreviations

NN Neural Network

DNN Deep Neural Network

RNN Recurrent Neural Network

ML Machine Learning

DL Deep Learning

BRF Benchmark Result File

HPO Hyperparameter Optimization

CNN Convolutional Neural Network

MSIS Mean Scaled Interval Score

MASE Mean Absolute Scaled Error

RMSE Root Mean Squared Error

MAPE Mean Absolute Percentage Error

List of Algorithms

1 Introduction

Time series forecasting, the ability to predict future trends from past data, is an important tool in many domains. For stock keeping businesses time series forecasting is used to plan how many items should be kept in stock to meet future demands and hydro electric power plants utilize timeseries forecasting to match power generation to demand [49, 46]. Predicting the future is however hard and predictions are only as good as the data and learning capabilities of the forecasting model used. Recently the Covid-19 pandemic had disastrous effects on society and time series forecasting was used to predict the spread of the virus so that governments, hospitals and companies could plan accordingly to minimize its effects. However, many of the forecasts were overestimating the spread of the virus which lead to both organizational and health issues for hospitals, personnel and patients. With better forecasting solutions which would be capable of generating probabilistic forecasts this issue could have been avoided [30].

1.1 Motivation

Improving forecasting accuracy is an active area of research, thus, new forecasting methods are continuously proposed [53, 49, 43, 44, 52]. As more forecasting methods are developed, these need to be compared in an accurate and reproducible way. The currently most popular method for comparing forecasting methods is through evaluating the algorithm on a couple of reference datasets [28].

Previous comparisons of forecasting algorithms have found subpar performance of machine learning based approaches both in terms of accuracy and resource consumption when compared to classical methods such as Arima, ETS and Theta [37, 38, 48]. However these comparisons are often made unfairly as the deep learning models tested are simple neural networks. More advanced deep learning methods such as DeepAR [53], DeepState[49], WaveNet [43] have been developed since, thus, there is a clear benefit of thoroughly evaluating these modern algorithms in a fair and accurate way.

Accurate and fair comparisons can however only be made if the algorithms being tested all have an equal opportunity to perform well. Equal opportunity would be if the algorithms are tuned to a similar degree or if equal dataset preprocessing has been done. Ensuring that different forecasting methods are given the chance to

perform optimally is hard and implementing a strategy for doing this is required. In addition to the algorithm and dataset tuning, the choice of error metric to use when comparing forecasting models is important since different error metrics have different benefits and drawbacks. Using a flawed error metric can invalidate a seemingly fair comparison as some metrics are unreliable for certain applications or datasets [23]. Implementing a strategy for when to use and not use certain error metrics is required in order to allow for fair comparisons.

Reproducibility of results is a core tenet of the scientific method, yet machine learning research is currently in a reproducibility crisis [9, 42]. Being able to reproduce results from forecasting algorithms is not always straightforward [38]. And robust methods for doing so is needed.

In other disciplines of machine learning benchmarking suites are common tools for performing automatic comparisons between different algorithms. Some examples of these are MLBench, MLPerf and DLBS [5, 41, 67]. In time series forecasting however, no established benchmarking suites exist, instead results from competitions such as the M-competitions are held as the reference which future papers compare to [37, 38, 39]. Creating a benchmark for the time series domain could improve the speed of innovation similarly to the M-competitions.

This thesis will focus on how fair, accurate and reproducible comparisons of modern forecasting methods can be made. Particular focus will be put on how comparisons can be made in a reproducible way for non-deterministic forecasting models. As part of this thesis, a system is implemented which automates the tuning and benchmarking of forecasting models. This system is then used to perform a large scale comparison of several modern forecasting algorithms over multiple datasets.

1.2 Challenges

When comparing forecasting algorithms, it is common practice to preprocess the datasets used for the comparisons. Similarly, the hyperparameters of the algorithms being compared is often optimized for the dataset it is being evaluated on. These steps are considered good practice as it allows algorithms to perform optimally on the datasets. However, if the method used to process the dataset is unclear in any way or if configuration details such as which hyperparameters used are missing, reproducing results becomes hard [38].

Some forecasting methods exhibit a non-deterministic behaviour where each subsequent run of the algorithm won't necessarily produce the same output. This is especially the case for deep learning algorithms as they are heavily relying on random processes such as dropout or random initialization of weights [65]. This makes it hard to reproduce results from these algorithms. Additionally, in the

field of forecasting, multiple different metrics are employed in order to quantify the error of forecasts. This is due to certain metrics being better suited for specific use cases. Often only a few of these metrics are used in papers when presenting new forecasting methods which makes reproducing results even harder.

Reproducible results imply that an algorithm needs to produce the same output no matter when someone wishes to reproduce them. It should not matter if it is shortly after the algorithm was presented or a long time thereafter. This introduces difficulties as some algorithms are continuously being improved upon by their makers. If any of these improvements would change the predictive power of the algorithm this would make it impossible to reproduce any results. These types of performance changes can be handled by automated tests which ensure that the accuracy of the algorithm remains. However, defining such tests is hard for non-deterministic output and therefore accuracy regressions can pass through undetected [31]. Changes in predictive performance can also stem from third party updates of dependencies. These things are hard to control and potential problems are hard to foresee ahead of time.

1.3 Research Question

This thesis has as an overarching goal to compare modern forecasting models. This question is complex in itself and can be separated into three sub-questions:

1. *Accuracy*: How to perform accurate comparisons between forecasting models?
2. *Fairness*: How can one fairly compare forecasting models?
3. *Reproducibility*: How to make comparisons reproducible?

1.4 Scope and Limitations

A requirement for performing fair comparisons is that the algorithms being compared are well implemented. As the goal of the thesis is to perform a large scale comparison of modern forecasting methods implementing multiple advanced algorithms is both time prohibitive and error prone. Thus, algorithms available in the open sourced forecasting library Gluon-TS will be used for this thesis. An additional benefit with Gluon-TS is that 21 popular datasets are available therein, as these are appropriately formatted for use with the algorithms already available in Gluon-TS these will be used in this thesis. Using these datasets saves time as data collection, cleaning and formatting is not required.

1.5 Contribution

The main contribution of this thesis is Crayon, an open source library for benchmarking deep learning models in a fair, accurate and reproducible way. Many deep learning based forecasting algorithms are non-deterministic in nature which makes reproducing results hard. Crayon solves this by ensuring that the distribution of error metrics are reproducible. With distributions of error metrics, quantifying how good the accuracy of an algorithm is becomes a problem. To handle this, Crayon implements a custom aggregation method of error distributions for ranking forecasting models against eachother. This aggregation method, named RMSE4D, is inspired by the Root Mean Squared Error (RMSE) [28] and benefits error distributions with low variance, thus this metric is biased towards consistency of error distributions rather than the best few runs of an algorithm.

This thesis also investigates the efficiency of using various hypothesis tests such as the T-test [51] and the Kolmogorov-Smirnov two sample test [40] to make non-deterministic output from forecasting models reproducible. Additionally, these tests are evaluated on forecasts generated by several modern forecasting solutions on several datasets. The practical benefits of applying hypothesis testing to verify distributions of forecasts is showcased on a modern forecasting algorithm trained on several real world datasets with multiple different hyperparameter configurations. Further, as a practical example this thesis shows how reproducibility by hypothesis testing can protect modern forecasting solutions from suffering accuracy regressions.

An additional contribution of this thesis is the analysis of 21 datasets with the purpose of identifying a representable subset to use when benchmarking forecasting methods.

As the final contribution of this thesis, a large empirical comparison is performed where the predictive performance of 13 modern forecasting algorithms is compared over four popular and publicly available datasets.

2 Scientific Background

This chapter introduces tools and concepts used to perform and evaluate time series forecasting and builds a basis for the work presented in Chapters 3, 4, 5 and 7. We start by establishing common features of time series forecasts, then in Section 2.2 methods of extracting such features from timeseries are introduced. In Section 2.3 methods and concepts used when forecasting on time series are shown and in Section 2.4 methods for evaluating the accuracy of forecasting methods are presented. An overview of how deep learning is applied to the field of time series forecasting is found in Section 2.5 along with specific issues related to deep learning methods. Other works relating to benchmarking machine learning methods and empricial comparisons of forecasting methods are discussed in Section 2.6 and in 2.7 three types of reproducibility are identified and requirements for fullfilling each are listed. Section 2.8 introduces the probabilistic forecasting framework Gluon-TS is and in Section 2.9 a tool for defining and executing large scale machine learning experiments is introduced. Finally in Section 2.10 non-parametric and parametric hypothesis tests are described along with four commonly used hypothesis tests.

2.1 Time Series

A time series is a set of data points with a clear ordering in time. Some examples are the exchange rate between two currencies over time or the electricity consumption of households, see Figures 1 and 2 [6].

Timeseries which increase or decrease in value over a longer period of time is said to have a upward or downward *trend*. If a timeseries has evenly distributed patterns such as the spikes in Figure 2 the timeseries has a seasonal component with a frequency equal to the distance between the spikes. For the electricity timeseries this is 24 as household electricity consumption follow the daily rythm of everyday life, i.e., more energy is used during the morning and evening with 24 hour intervals. If the timeseries exhibit spikes without a clear frequency, these are knowns as *cycles* [8]. The data which cannot be described by these features is known as the *remainder* [28]. Not all timeseries exhibit all of these patterns, for example, the timeseries in Figure 2 does not display a trend.

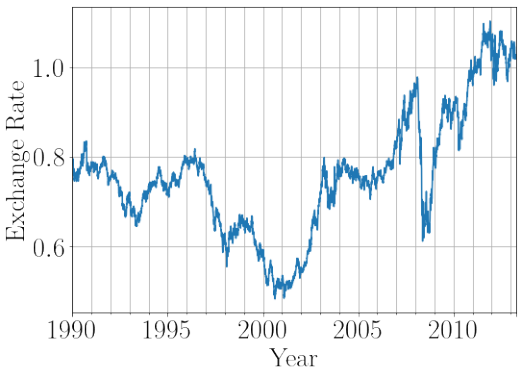

Figure 1: Exchange rate of two currencies from 1990 to 2013.

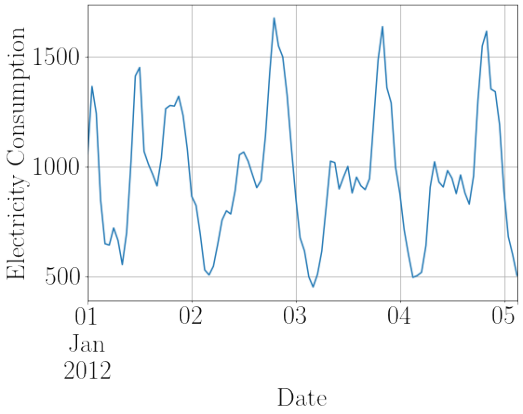

Figure 2: Household electricity consumption.

2.2 Time Series Decomposition

Timeseries decomposition is the act of extracting features such as the seasonality and trend from timeseries data. This is often done in order to gain understanding of the data [28]. While several methods of performing time series decomposition exists, one of the most powerful methods commonly used is STL decomposition [28]. After a timeseries has been split into its constituents one can quantify the strength of the trend and seasonality apparent in the data. This is done through compairing the variances of the seasonality S and trend T with that of the residual R. The strength of the trend F_t is then calculated according to Equation 3 and the strength of the seasonality F_s is calculated according to Equation 4.

$$F_t = max(0, 1 - \frac{Var(R)}{Var(T + R)})$$

Figure 3: Equation for calculating the strength of the trend of a timeseries.

$$F_s = max(0, 1 - \frac{Var(R)}{Var(S + R)})$$

Figure 4: Equation for calculating the strength of the seasonality of a timeseries.

2.3 Time Series Forecasting

Time series forecasting is the art of predicting future trends of time series based on past observations. It is a key technology within many fields and is fundamental to the business decision process for many companies. Being able to interpret time-series in order to predict future trends makes it possible to plan for the future. For example power generation companies can adjust the amount of electricity which should be generated to match the expected demand and banks could make informed decisions about possible investment opportunities. There are many methods of generating such predictions, ranging from the use of simple heuristics to complex machine learning methods [28]. An overview of how to use deep learning to generate such forecasts is presented in Section 2.5 and several state of the art forecasting models are presented in Section 2.8.1.

Time series forecasting differs from common machine learning tasks such as image recognition or language processing in that predictions are not binary. More specifically, in time series forecasting a prediction is expected to be a close estimate of future values. Thus, quantifying the magnitude of the error from the ground truth is of importance in order to properly evaluate the accuracy of a forecasting model. This is commonly done by calculating an error metric, the choice of which varies depending on what is being forecasted and on preference of the forecasting practitioner. [28, 27, 71] In Section 2.4.1 some such error metrics are presented.

Forecasts often comes in one of two forms; probabilistic forecasts or point forecasts. A point forecast is a forecast which consist of only one estimation of future values. This estimation could be either a single point in time or a series of points for several different timesteps. In Figure 5 an example of a point forecast for several timesteps can be seen. Point forecasts has the disadvantage that they do not confer how accurate they are which may lead to prectitioners being overly confident in the forecasts. This has historically caused several issues and more recently with covid-19 where over relying on point forecasts led to overly drastic measures for governments and hospitals [30]. Probabilistic forecasts such as the one in Figure 6 includes this information by generating prediction intervals of the forecasts.

Many methods exist for generating forecasts on timeseries data, two common groups of these are machine learning based methods and trivial methods. A trivial forecasting method would be, for example, to predict that the next value should be the same as the last recently observed value or the average of the last N observations. For certain timeseries, trivial methods perform competitively and as such these are often held as a baseline to which new forecasting methods are being compared too. Some more complex methods include those based on regression analysis such as linear regression or autoregressive models such as ARIMA [28]. Autoregressive models are a subset of regression models where the temporal or-

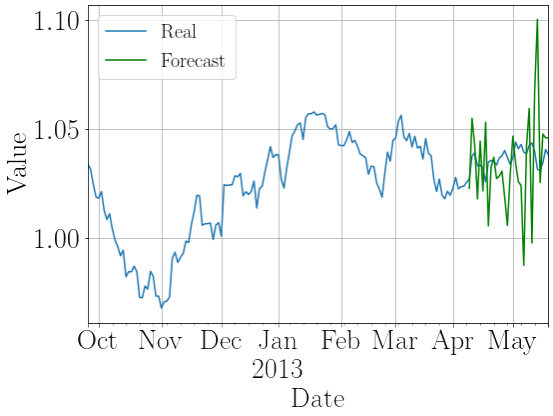

Figure 5: Point forecast

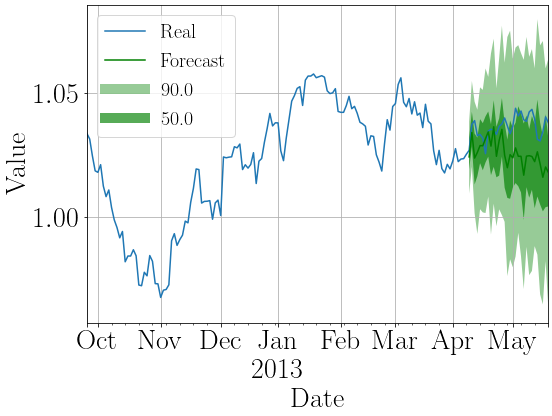

Figure 6: Probability forecast

dering of the datapoints matter. I.e. autoregressive models use past observations to predict future values.

Lately, inspired by the successful use of neural networks in other domains, several neural network based approaches for time series forecasting have been introduced. Early implementations such as simple multi-layered perceptrons did not perform competitively with the more classical approaches such ARIMA. However modern deep learning models such as DeepAR [53] and MQCNN [70] have reported highly competitive accuracies.

Forecasting models for forecasting can also be split into further sub families such as whether they are local or global models. Local models are trained on individual time series while global models are trained on all time series in the training set. Essentially for a dataset with N timeseries, if using local models, one will have N individually trained models. If the algorithm would be a global model then only a single model would be used for all N timeseries in the dataset. Generally speaking, local models performs well for timeseries with much historical data. However for short or new timeseries, local models often perform poorly due to the inability to train the model on sufficiently large amounts of data points. This is known as the cold start problem. Since global models train on the entire dataset across all timeseries, the length of any individual timeseries has less of an impact. I.e. generally global models suffer less from the cold start issue as the global model can use data from other timeseries as a basis for its predictions [69].

So far, we have seen the time series forecasting problem as generating a forecast from a single timeseries for example predicting future temperatures based on previously recorded temperatures. Predicting using singular timeseries such as this is known as univariate forecasting. Another family of forecasting solutions leverage related timeseries when making predictions. For example temperatures

may depend on cloud coverage or the hours of sunlight. Algorithm which leverage information from related timseries are known as multivariate algorithms. The benefits of multivariate forecasting is that trends or patterns can be identified using these related timeseries in order to improve the accuracy for the target variable. For example, the temperature is higher whenever there is a large amou8nt of sun hours and no clouds.

2.4 Evaluating Forecasting Performance

If a model generates forecasts it is important to be able to quantify how good or bad its predictions are. This is generally done via a process called backtesting. When backtesting, first the algorithm is trained on a dataset, thereafter, the trained algorithm generates predictions on a test dataset. The generated predictions from the algorithm are then compared to the ground truth and a suitable error metric quantifying how wrong the prediction was from the expected value is calculated.

In other machine learning domains such as image recognition, the datasets used for training the models are disjoint from the test datasets. However, in the domain of time series forecasting this is not the case. Generally the train set contains the same time series as in the test set except for the last couple of data points for each time series. The number of removed data points is commonly the number of timesteps that a model should predict [28]. In Figure 7 a train test split is presented of a time series with six data points. In this example, a forecasting algorithm should predict one timesteps into the future, thus, the last datapoint would be removed and the train set would contain five data points while the test series would contain six.

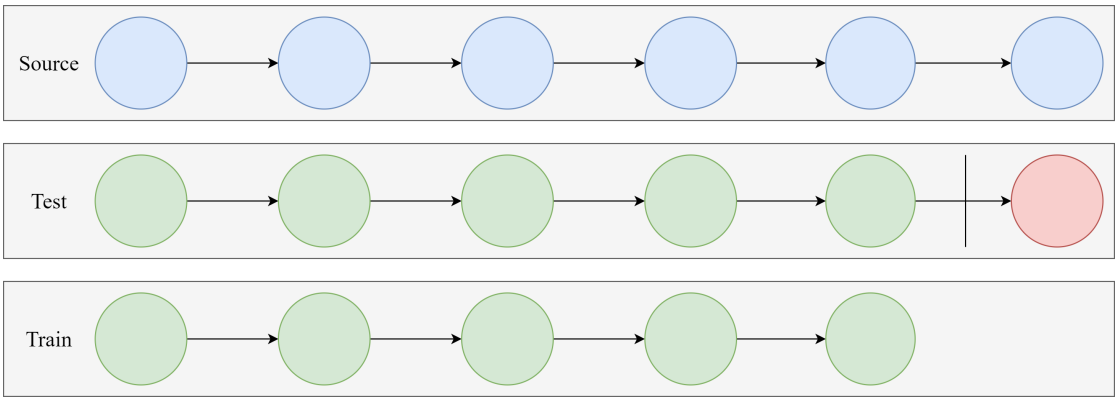

Figure 7: An example train test split of a time series with six datapoints. The red circle is the datapoint which prediction will be compared to.

A popular method of increasing the amount of test data is to make the test

dataset contain multiple copies of each time series where each one is successively shorter. The train dataset would contain the shortest of the truncated time series truncated one additional time. This process is called K-fold cross validation but is also known as forecasting on a rolling origin or time series cross validation. Performing a 3-fold cross validation on a dataset containing one time series of length six timesteps would result in a dataset containing three time series. An example of this is shown in Figure 8. The advantage of the larger test dataset generated by this approach is that the performance of a model can be more accurately determined as it is excercised on more unseen data [28].

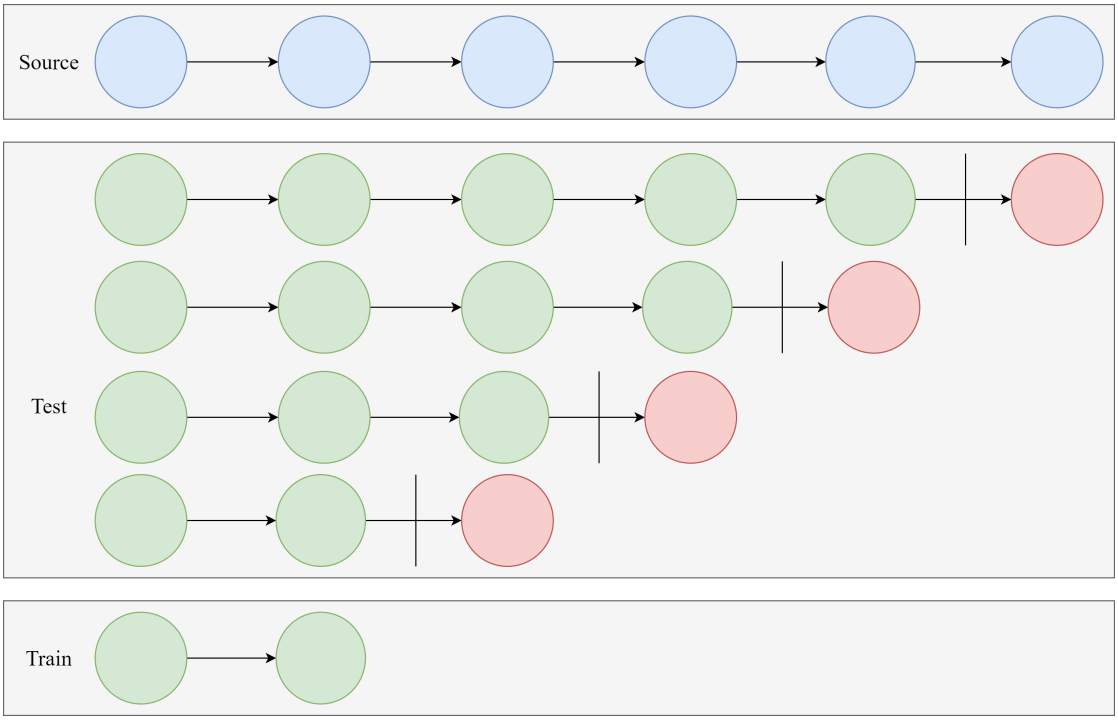

Figure 8: Example of a four fold cross validation dataset split. The red circles are the datapoints which predictions will be compaired to. Worth noting is that the train dataset here only has one timeseries of length two, i.e. one less than the shortest timeseries in the test dataset. This example assumes a prediction length for the dataset of one datapoint.

2.4.1 Error Metrics

When a forecasting model generates predictions on a timeseries, one needs to be able to quantify the error of the prediction from true observed values. In order to do this many different error metrics exists each with its own benefits

and drawbacks. Often the choice of metric is dependent on data, and business application where the prediction will be used. In this section some of the error metrics which are commonly occurring in literature are presented. These metrics are calculated automatically when performing a backtest in Gluon-TS, see 2.8 for details about this process. Note that only a subset of the metrics available in Gluon-TS is presented here to form a basis for subsequent chapters.

Error metrics for forecasting can generally be split into three different categories: scale-dependent, scaled and percentage based [28]. Within each of these categories multiple different metrics exists. The remainder of this section describes some common error metrics within each category along with pros and cons and how they are calculated. For the rest of the thesis, a forecast is referred to as $\hat{Y}$ and the expected values are referred to as Y.

Absolute Error

The absolute error is calculated as the distance between the predicted value and the ground truth. An absolute error is positive and a value of 0 is optimal. It is an intuitively simple error, thus it can easily be understood and discussed when comparing algorithms.

However intuitive the absolute error is, there is a a key issue with this error and that is that it is scale dependent, i.e., the greater the values of the dataset being predicted on, the larger the absolute error will be. This makes it impossible to compare the accuracy of an algorithm across different datasets unless they have the same scale. [28] Furthermore, scale dependent metrics such as these may not be suitable even for comparing between timeseries within a single dataset. Imagine a dataset such as the electricity dataset (see Table 3) where each timeseries is the electricity consumption from a household. One household may contain many people and appliances while another may be a factory full of equipment with high power demands. These two timeseries would have two very different scales thus making it hard to compare the accuracy of the predictions.

$$AbsoluteError = \sum |Y - \hat{Y}|$$

Figure 9: Equation for calculating the absolute error

Root Mean Squared Error - RMSE

The root mean squared error is another scale dependent error such as the absolute error. RMSE is calculated by taking the square root of the mean of the squared

error. The RMSE thus punishes larger errors more than smaller errors. The RMSE is more complicated to interpret than for example the Absolute Error due to the non-linear nature, despite this, RMSE is widely used in practise [28, 3].

$$RMSE = \sqrt{mean((Y - \hat{Y})^2)}$$

Figure 10: Equation for calculating RMSE

Mean Absolute Percentage Error - MAPE

Percentage based errors such as the Mean Average Percentage Error normalizes the errors between 0 and 1 thus MAPE can be compared across datasets with different scales. While this is beneficial, MAPE suffer from other issues. For example MAPE becomes infinite or undefined if the ground truth is 0 for any parts of the prediction. Further issues exists for example that it penalizes negative errors more than positive errors. These issues has lead to variations of MAPE to be created such as Symmetric MAPE [28, 3].

$$MAPE = mean(\frac{|Y - \hat{Y}|}{|Y|}))$$

Figure 11: Equation for calculating MAPE

Mean Absolute Scaled Error - MASE

MASE is a metric which takes the average error across all timeseries and scale is based on the error from a seasonal naive forecaster such as the one detailed in Section 2.8.1. Scaling the forecasting error in this way causes the MASE metric to be scale independent which makes it suitable when comparing across datasets. A MASE of 1 corresponds to that the algorithm under test is equally good as the naive model. A MASE below 1 means that the current model performs better than the naive model while a MASE above 1 means that the model performs worse than the naive baseline model [28, 3].

In equation 12 the denominator is the mean error of a seasonal naive model. Y_t means the value at time t and Y_{t-m} means the value of the timeseries at the

$$MASE = \frac{mean(|Y - \hat{Y}|)}{mean(|Y_t - Y_{t-m}|)}$$

Figure 12: Equation for calculating MASE

previous period, i.e. if the expected seasonality is 24h Y_{t-m} would become the value 24 timesteps previously.

In certain scenarios such as if the target time series is constant, the value of the previous period Y_{t-m}, is equal to the value at the current timepoint Y_t. This causes the seasonal naive model in the divisor to a perfect prediction with zero error. When this happens, a division by zero occur which causes MASE to tend towards infinity. This is a major drawback of the MASE metric which renders it unusable for certain datasets.

Mean Scaled Interval Score - MSIS

Since probabilistic forecasts do not generate point predictions but prediction intervals these can also be scored in regards to how often these contain the true values Y. The MSIS metric penalizes wide prediction intervals in two ways. First, the absolute width of the prediction intervals is penalized. Secondly, the distance from the edge of the prediction interval from the expected value Y is penalized. Thus smaller more accurate and consistent prediction intervals are encouraged [38]. In Equation 13 $\hat{Y}_u$ is the upper limit of the forecast prediction interval and $\hat{Y}_l$ is the lower boundrary. The α parameter is the significance level used, this is commonly set to 0.05 to evaluate the 95% prediction intervals [38, 3]. The $I(x)$ function is the indicator function which is 1 if $\hat{Y}$ is within the prediction interval and 0 otherwise. The denominator used in Equation 13 is the mean seasonal error, i.e. the same denominator for the MASE metric. This division makes the MSIS metric scale independent [3, 38].

$$msis = \frac{mean(\hat{Y}_u - \hat{Y}_l + \frac{2(\hat{Y}_l - Y)}{\alpha} I(Y < \hat{Y}_l) + \frac{2(Y - \hat{Y}_u)}{\alpha} I(Y > \hat{Y}_u))}{mean(|Y_t - Y_{t-m}]|))}$$

Figure 13: Equation for calculating MSIS

2.5 Deep Learning Forecasting Methods

Neural Networks (NN) have in recent years been successfully applied in many different domains such as image recognition and natural language processing. However, within the field of forecasting NN based approaches has performed subpar compared to more traditional statistical methods [37, 38, 44, 48]. This relative inefficiency between NN based approaches and classical methods and the success of NN in other domains has resulted in much research being made in improving tools and NN based models for forecasting [6].

A simple neural network, similar in architecture to the network described in Section 2.8.1 is presented in Figure 14.

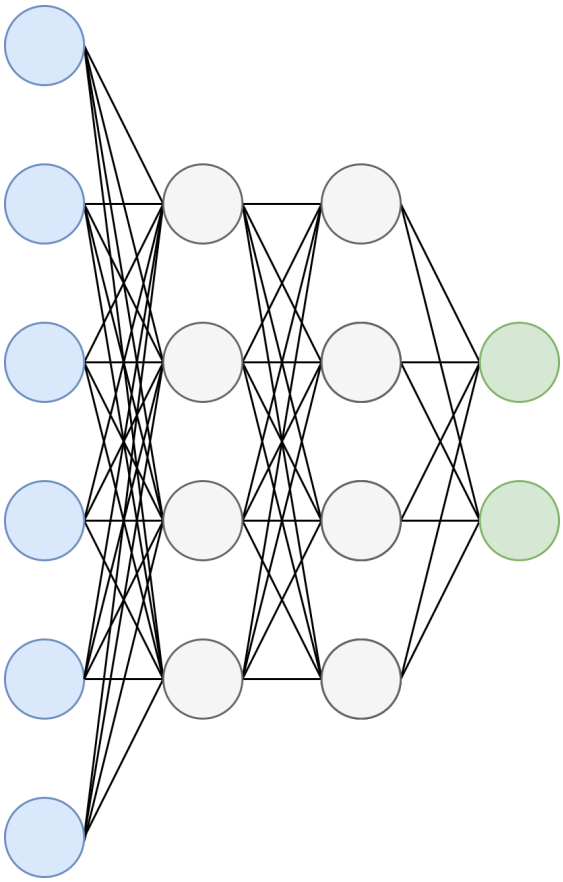

Figure 14: A simple neural network for forecasting based on the SimpleFeedForwardEstimator in Section 2.8.1.

This very small network consists of four fully connected layers of different sizes with six input nodes, two fully connected hidden layers with four nodes each, and an output layer of two nodes. In this network, the input layer (marked in blue) has the same amount of nodes as the number of timepoints in the past which the network should make use of. This is for the remainder of the thesis referred to as the *context length* of the network. The output nodes (in green) for this network produce a prediction of two timesteps. Thus this network has a *prediction length* of two timesteps. The context length and the desired prediction length are often exposed as hyperparameters for time series forecasting algorithms since they vary between datasets.

An issue with the simple network in Figure 14 is that each timestep in the forecast is not influenced by the previous one. i.e. the prediction at timestep N does not depend on the prediction at timestep $N-1$. In real life scenarios, timeseries data often has a temporal dependency, for example, the temperature two days from now is affected by the temperature of tomorrow. This dependency is lost in simple neural network architectures such as this one. An alternative neural network architecture which can make use of temporal effects is known as a Recurrent Neural Network (RNN).

RNNs reuse the output of their nodes as inputs for the next iteration of the algorithm. This recurrent link causes previously seen values in the sequence to be represented in a hidden internal state. While this makes RNNs suited for use with sequential data such as time series, they become hard to train successfully as long term dependencies are lost [10]. One of the most successful techniques for use in RNNs to solve this is the Long Short-Term Memory (LSTM) node architecture. LSTM nodes enables RNNs to remember previously seen data for longer and forget or ignore parts of it [60].

Some issues with NN based approaches is that they require a large amount of data to be trained on properly. This is due to the fact that NNs are prone to overfit which makes them unable to generalize well to new data [65]. Despite this, NN has several advantages over classical approaches such as the capability to learn non-linear patterns from the data. Additionally, NN models are capable of cross learning between related time series without the need of manual feature extraction [62].

2.5.1 Sources of Randomness

Neural networks heavily rely on random processess to function and thus randomnes is introduced at several points in a forecasting workflow. Some randomness can be introduced whenever an neural network is initialized as the weights and biases of a networks often are set randomly. Similarly, when training a neural network randomness can be introduced if optimization techniques such as dropout is used

as dropout randomly sets weights in the network to 0 at certain points in the training loop which causes the algorithm to be less prone of overfitting the data [65]. Even the method used for training neural networks, stochastic gradient descent introduces randomness.

Deep NNs are notoriously slow to train when compared to many other methods, thus lowering the time spent for training and inference is often done by parallelizing algorithms such that they run on several threads on a CPU or on hardware accelerators such as GPUs, FPGAs or ASICs. The execution speed of a forecasting model is thus highly dependent on what hardware the models are running on, however the hardware configuration also impacts the variance of the accuracy of ML models [72]. Parallelization of an algorithm often introduces a larger amount of data shuffling which impacts the rate of convergence of an algorithm. Parallelizing a forecasting model may make it unstable so that it cannot converge as will be shown in chapter 7. Due to this it is important that the hardware used when training and tuning forecasting models is presented in detail so that results can be properly reproduced [47].

A common method for reducing the randomness associated with ML models is to fix the seed used by the random number generator to a known number. While this removes randomness from for example random initializations or from backpropagation [9], other sources of randomness such as that introduced by the hardware or the OS remain. Furthermore, fixing the seed can be seen as an additional hyperparameter which needs to be tuned as it directly impacts the predictive performance of forecasting methods. While there are some major benefits of fixing the seed, there are also drawbacks. For example,luck becomes a factor in what accuracy can be expected for an algorithm. I.e. the accuracy of a model may seem highly competitive while if another seed would be used the accuracy of the model could change significantly [9]. In real life usecases such as when forecasting are used in companies or organizations, setting the seed may be beneficial or even necessary in order to optimize its performance for an individual task. In research however, setting the seed without reporting what it was can bias results and make them technically irreproducible [9, 47, 11].

One method of lowering the variance of machine learning methods is to use ensambles of forecasting methods as per the Bootstrap AGGregatING (Bagging) technique [12]. An ensamble is a set of multiple different ML models with slightly different configurations which are all trained in parallel.Bagging extends this such that a the input dataset is randomly sampled to create multiple independent datasets. Each model is then trained on one of these datasets and their forecasts are then combined via some voting or weighting scheme [12]. While bagging is often associated with decision trees they are also suitable for use with deep learning forecasting models such as the N-BEATS Ensamble Estimator discussed

in Section 2.8.1.

K-fold cross validation which was discussed in Section 2.4 can also lower the variance of forecasting algorithms as it introduces multiple different train-test splits from a single dataset [12].

2.6 Benchmarking

Backtesting is a fundamental part of benchmarking, however benchmarking is also concerned with ensuring that different models can be fairly and reproducibly compared to each other [26]. In other domains of machine learning several benchmarking solutions exists, most notably MLPerf [41], MLBench [5], DLBS [67] and UCR [13]. Despite the success of benchmarks in other ML domains no established commonly used benchmarking suite exists in the domain of time series forecasting [26].

Attempts has been made to create such a benchmark, one such example is *Libra* [8] which uses custom datasets containing timeseries sampled from various well known datasets. The sampling methodology used in Libra was aimed to create datasets with a high amount of diversity between the timeseries. Due to this diversity, the datasets in Libra are not suitable for global forecasting models as they are highly unrelated. In a nutshell, Libra focuses on benchmarking the accuracy and time-to-result for univariate forecasts generated by local models.

Instead of using benchmarking suites when comparing forecasting models, new forecasting solutions are instead pitted against each other in forecasting competitions. The Makridakis Competitions organized by Makridakis Spyros et al. are the most well known of these and has at the time of writing undergone five iterations; the M1 in 1983 [36], M2 in 1982 [35], M3 in 2000 [37], M4 in 2018 [38] and the M5 competition in 2020 [39]. These competitions have been highly impactful for the field of forecasting and has been the norm to which researchers compare too. However it was not until the M3 competition that NN based forecasters were included. These performed poorly compared to the non-NN algorithms which can partly be attributed to the limited size of the M3 dataset of 3003 timeseries [37]. This shortcoming of the M3 dataset lead to the creation of the M4 competition which had a similar but much larger dataset containing 100 000 timeseries [38]. Despite this, the findings of the M4 competition was inline with that of the M3 competition in that approaches purely using NN performed worse than even simple models. However, the M4 competition winner was a hybrid approach which combined RNNs with classical models. The latest competition, the M5, was focused on retail sales forecasting and found for the first time that ML based solutions, primarily Gradient Boosting Trees but also NN approaches such as DeepAR outperformed classical statistical methods [39].

Name	TS	Integration	DS	Focus	Info
Libra	X	R	X	A, S	Diverse datasets for univariate local models
UCR	X	CSV	X	A	Dataset repository with reference accuracies.
MLPerf		Docker, CLI		S	Benchmarking for hardware and ML frameworks
PMLB	X	Python, R	X	-	Dataset repository with 298 datasets for regression and classification.
MLBench		Kubernetes		S	For distributed ML frameworks
DLBS		Docker, Python		S	Benchmarking for hardware and ML frameworks

Table 1: Overview of six benchmarking suites for ML, the column marked TS depicts whether they support time series forecasting and the DS column is whether they offer custom datasets. A is shorthand for Accuracy and S for Speed.

In 2015 a thesis comparing forecasting solutions was published with the topic: *Benchmarking of Classical and Machine-Learning Algorithms (with special emphasis on Bagging and Boosting Approaches) for Time Series Forecasting* [48]. This thesis put the focus on performing a thorough dataset analysis and evaluating 14 algorithms on three real world datasets; Tourism, M3 and the NN5 [2]. Each algorithm was evaluated on multiple versions of each of these datasets with different preprocessing applied to them. In addition to evaluating the algorithms on the three real life datasets, a simulation study on artificial data was conducted. The purpose of this was to identify strengths and weaknesses of the algorithms for timeseries with simple characteristics such as only having trend or seasonality.

It is unclear whether time series cross validation was used as well as how many times each algorithm was executed. Further, as the title suggests there was little emphasis on deep learning methods and only a very simple neural network, similar to the one presented in Figure 14 was used. This neural network performed worse than the naive approach for all datasets except for on NN5 [48].

2.7 Reproducibility

The ability to reproduce scientific results within the field of machine learning has been a hot topic in recent years with major conferences such as NeurIPS announcing dedicated reproducibility programs [47]. Certain fields of ML research such as within healthcare is reportedly in a *reproducibility crisis* [9, 42]. Surveys conducted in 2018 reported that only 63% of 255 ML publications could be reproduced and only 4% of them could be reproduced without the original authors assistance [47]. Another survey in Nature sent out to more than 1500 scientists from various diciplines in 2016 reported that 50% researchers failed in reproducing even their own experiments [7]. Also Makridakis et al., the organizers of the M-competitions, urged researchers to improve the reproducibility of their work. The following is a quote from their paper summarizing the M4 competition: *"We believe that there is an urgent need for the results of important ML studies claiming superior forecasting performance to be replicated/reproduced, and therefore call for such studies to make their data and forecasting algorithms publically available, which is not currently the case"* [38].

There are multiple methods of defining reproducibility. Pineau et al. defined reproducibility as being able as *re-doing an experiment using the same data and same analytical tools* [47]. This definition aligns with that of Beam et al. however, they point out that reproducibility is not concerned with the validity of the claims of a paper, only that practical results can be recreated [9]. McDermott et al. further specifies the term reproducibility into three parts *technical*, *statistical* and *conceptual*. For technical reproducibility, the code and datasets used should be made publically available and in order to be statistically reproducible, the variance of the results should be published. To be conceptually reprodicble, it was important that multiple complimentary datasets should be used for the comparisons [42].

In addition to the requirement that code and datasets should be made available, Pineau et al. suggested that standardized tools for supporting reproducibility should be developed and that any metrics used should be sufficiently specified. Furthermore it was argued that encapsulation tools such as Docker should be used as these would remove OS and dependency related issues [47]. This would also resolve issues pertaining to different versions of code being used [9]. The random seed which was discussed in Section 2.5.1 is also an important value which should be reported as it can drastically affect the performance, especially for neural network based models [9, 11]. Increasing the use of statistical tests is another key aspect for improving the reproducibility [42]. In addition to this forecasting models should be executed multiple times with as much variation as possible between each run to further improve the accuracy and the statistical reproducibility of the results [11].

In table 2 the requirements this section identified for achieving true reproducibility is presented. To summarize, technical reproducibility is achieved by making assets such as code, datasets, seed and Docker images publically available. Variance of error distributions and the use of statistical tests are required for statistical reproducibility while conceptual reproducibility requires complimentary datasets to be used in the comparisons.

Type of reproducibility	Requirements
Technical	Code and datasets should be publically available. If random seed is set it should be reported. Specific versions of code should be detailed and assets such as Docker containers provided.
Statistical	Variance of results should be published and statistical tests should be used.
Conceptual	Multiple complimentary datasets should be used for the comparison.

Table 2: Requirements for making ML workloads reproducible.

2.8 Gluon-TS

Gluon-TS is an open source library which includes several deep learning algorithms, datasets and tools for building and evaluating deep learning forecasting methods [4, 6, 3]. In Gluon-TS, the *Estimator* class corresponds to the implementation of a forecasting method. An *Estimator* can be trained on a dataset in order to generate a *Predictor* object. This *Predictor* can then ingest timeseries in order to produce forecasts for them.

In order to make it easy to evaluate the performance of an algorithm, Gluon-TS offers the *backtest_metrics* function. This function trains an *Estimator* on a train dataset, the generated *Predictor* is then used to generate predictions on a test dataset. Each of the timeseries in the test dataset will have the final N datapoints removed prior to passing them to the *Predictor*. The *Predictor* then takes the remaining time series data and produces a prediction of the same length N. Thereafter, the previously truncated datapoints are compared with the generated forecast in order to calculate error of the prediction from the ground truth [3]. There are multiple ways to calculate this error some of these are presented in detail in Section 2.4.1.

Gluon-TS also contains Dockerfiles which makes it easy to create Docker images with installations of Gluon-TS inside. These images are compatible with AWS

SageMaker which allows them to be executed both in the cloud as well as locally. These images executes the *backtest_metrics* function of Gluon-TS when run and can thus be used to generate error metrics for any *Estimator* class in Gluon-TS [3].

2.8.1 Algorithms

Gluon-TS offers several forecasting models which can be used to generate predictions on timeseries data. Most of these algorithms are presented below. Only algorithms which were well documented in Gluon-TS or which I could explain by sifting through the source code are presented below.

CanonicalRNNEstimator

The CanonicalRNNEstimator is a bare bones RNN with a single layer of LSTM cells and is capable of producing probabilistic forecasts [3].

DeepAREstimator

DeepAR is a RNN which produces probabilistic forecasts and was presented in *DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks* [53]. The model learns a global representation across all timeseries in a dataset and learns to identify seasonal behaviours in the data. DeepAR automatically adds certain meta information to the timeseries such as *day-of-the-week*, *hour-of-the-day*, *week-of-the-year* and *month-of-the-year*. Normally this type of meta information is manually added by the data scientist using the model, thus automating this procedure minimizes manual feature engineering. Another feature of DeepAR is the possibility to choose a likelihood distribution according to the data that it is to be trained on. Two different distributions are used by the original authors, the Gaussian Likelihood for real valued data and the negative binomial likelihood for postive count-data.

DeepFactorEstimator

The DeepFactorEstimator was presented in the paper *Deep Factors for Forecasting* in 2019 and consists of two main parts, a global and a local model [69]. The global model is a deep neural network (DNN) which is trained across all timeseries in order to capture complex non-linear patterns between the timeseries. The local model is a simpler model which is meant to capture local trends and patterns of individual timeseries. This hybrid architecture is thus able to leverage the DNN capability of learning complex patterns as well as having the computational efficiency of local models. Three different versions of the DeepFactorEstimator is presented in the

original paper, the one implemented in Gluon-TS is the DF-RNN version. This version uses a second RNN to model the local timeseries. The DF-RNN presented in the paper performs better than Prophet and MQ-RNN on the Electricity, Taxi, Traffic and the Uber dataset both for short and long term forecasts. DeepAR performed worse than DF-RNN on average but was more accurate on the short term forecasts for the Uber dataset. The Uber dataset is not available as part of Gluon-TS however it is mentioned here for completeness, the other datasets are detailed in Table 3.

DeepStateEstimator

The DeepStateEstimator combines linear state space models for individual time-series with a jointly learned RNN which is taught how to set the parameters for the linear state space models. This has the benefit that the labour intensive tuning of multiple state space models is done automatically by the RNN without sacrificing performance. This approach was shown to perform better than DeepAR on the electricity dataset and on 4 out of 6 metrics on the traffic dataset. Furthermore, this approach outperformed ARIMA and ETS on all datasets [49].

GaussianProcessEstimator

The GaussianProcessEstimator is a local model where each timeseries in the dataset is modeled by a single gaussian process. The specific kernel used in the gaussian process can be specified as a parameter [4].

GPVAREstimator and DeepVAREstimator

The GPVAREstimator is a RNN model for handling multivariate timeseries which uses a gaussian copula technique for automatic data transformation and scaling. It utilizes a dimensionality reduction technique which minimizes the complexity of calculating a covariance matrix from $O(n^2)$ to $O(n)$. This allows much larger multivariate timeseries datasets to be used with it than what was previously possible. Additionally, the GPVAREstimator was compared against other multivariate forecasting algorithms, amongst them was a multivariate version of the DeepAR algorithm. This augmented version of DeepAR is implemented in Gluon-TS under the name DeepVAREstimator [52].

LSTNetEstimator

The LSTNetEstimator is a hybrid approach where long term patterns of the data is captured by a neural network architecture, these long term patterns are combined with a classical autoregressive model when generating predictions. The neural

network architecture consists of four parts, a CNN, a RNN a novel RNN-skip layer and a fully connected layer. The RNN-skip layer makes up for the incapability of the LSTM cells in a RNN to remember very long term information by only updating the cells periodically [33].

N-BEATSEstimator and N-BEATSEnsembleEstimator

In the paper *N-BEATS: Neural basis expansion analysis for interpretable time series forecasting* [44] a univariate deep learning model for forecasting named N-BEATS is presented. The authors of N-BEATS expressed that their goal with this algorithm was to disprove the notion that deep learning models had inferior performance compared to classical models. The authors reported a superior accuracy of N-BEATS model over all other models it was compared to.

The N-BEATS algorithm is an ensemble of 180 neural networks which were all trained to optimize for different metrics such as MASE, sMASE, MAPE and six different context lengths. Furthermore, bagging was used by running multiple runs of the algorithms with random initializations of the networks. A prediction of the N-BEATS model is the median value of the predictions of the algorithms in the ensemble.

Each model in the ensemble consists of multiple fully connected layers chained together to form blocks of networks. Each block is in turn chained together in order to form a stack. These stacks are also chained in order to form the final network.

Gluon-TS implements the N-BEATS model as the N-BEATSEnsambleEstimator and the smaller algorithms in the ensemble as the N-BEATSEstimator class. There are some differences between the implementation in the original paper and in Gluon-TS. Specifically, the training data is sampled differently in Gluon-TS than how it was in the source paper [4].

Naive2Predictor

The Naive2Predictor is a Gluon-TS implementation of the Naïve 2 forecasting method used as a reference in the m4 competition [38]. The Naive2Predictor predicts the future values to be that of the last datapoint in the timeseries adjusted by some seasonality. In Gluon-TS this seasonality can be deduced from the frequency of the data or by passing a custom seasonality via the *season_length* parameter [4].

NPTSPredictor

The NPTSPredictor is not a NN instead it predicts future values by sampling from previous data in the timeseries. The way that the samples are selected from

the previous data can be modified via the hyperparameters. One can sample uniformly across all previous values in the timeseries or bias the sampling to more often sample from more recent datapoints depending on which kernel is used [4].

ProphetPredictor

Prophet is a nonlinear regression model developed at Facebook which frames the timeseries forecasting problem as a curve fitting exercise [28]. This is different from many other forecasting models for timeseries which leverage the temporal aspect of timeseries in order to generate forecasts. Prophet was created with three goals in mind; it should be easy to use for people without much knowledge about timeseries methods, it should work for many different forecasting tasks which may have distinct features. Finally it should contain logic which makes it easy to identify how good the generated forecasts of Prophet are [66].

RForecastPredictor

The RForecastPredictor allows users of Gluon-TS to use forecasters from the popular Rlang forecasting package *forecast* inside of Gluon-TS. The *forecast* package contains several popular classical forecasting methods such as ETS and Theta. ETS is a family of forecasting methods which generates forecasts by computing a weighted average over all previously seen datapoints in the time series. Multiple versions of ETS exists with different weighting schemes and the *ETS* method in *forecast* automatically chooses the most suitable for the timeseries being forecast on [1]. Another name for ETS is exponential smoothing [28].

Theta is a univariate forecaster comparable to simple exponential smoothing with drift and is in the *forecast* package named *thetaf* [29].

The forecaster to use from the *forecast* package can be selected by passing the name of the method as a hyperparameter [4, 1].

SeasonalNaivePredictor

The SeasonalNaivePredictor is a naive model which predicts the future value to be the same as the value of the previous season. This is a very simple model however an example explains its functionality best. If I want to use the SeasonalNaivePredictor to forecast the average temperature of June. The SeasonalNaivePredictor would return the average temperature for last year in June. In Gluon-TS, if not enough data exists (i.e. no data for last year in June) the mean of all the data in the timeseries is returned [4, 28]

MQCNNEstimator & MQRNNEstimator

The Gluon-TS seq2seq package contains two forecasting algorithms, MQ-CNN and MQ-RNN. These are based on the MQ framework described in *A Multi-Horizon Quantile Recurrent Forecaster* [70]. The MQ framework is based on the sequence too sequence (Seq2Seq) architecture which consists of an encoder and a decoder network [24]. A Seq2Seq architecture encodes the training data into a hidden state which the decoder network then decompresses. Normally in Seq2Seq architectures, the RNN models tend to accumulate errors as the forecasts of an RNN for a timepoint t will be reused in order to generate a forecast for time $t+1$. By instead training the model to generate multiple point forecast for each timepoint in the horizon one wishes to forecast on, the errors tend to grow smaller. This is called *Direct Multi-Horizon Forecasting* and it is one of the changes introduced as part of the MQ framework. Further, the MQ framework allows for different encoders to used. Two of these are implemented in Gluon-TS, one with a CNN and one with a RNN. These are known as the the MQCNNEstimator and the MQRNNEstimator respectively [4].

SimpleFeedForwardEstimator

The SimpleFeedForwardEstimator in Gluon-TS is a traditional Multi Layer Perceptron (MLP) capable of generating probabilistic forecasts. The size of the network can be adjusted based on user parameters. Per default it has two densely connected layers containing 40 nodes in the input layer and 40 times the desired prediction length as the number of cells in the hidden layer.

The network has an additional layer which allows it to generate probability based predictions instead of only point predictions. This layer consists of a number of sublayers equal to the desired prediction length. Each of these layers consists of 40 nodes [3].

TransformerEstimator

The transformer is a Seq2Seq model which replaces the classical RNN encoder and decoders in a Seq2Seq model with more easily parallelizable NN components. A RNN requires data to be passed sequentially in order to learn dependencies between datapoints. This introduces a bottleneck and makes RNNs harder to train efficiently on modern hardware accelerators such as GPUs. The Transformer architecture presented in *Attention is All you Need* [68] alliviates this by leveraging parallelizable architectures such as feed forward networks as well as making heavy use of attention. Attention means that the architecture automatically can identify which parts of the input data is most relevant to the value we are trying to

predict[68]. I.e. for the next value we are predicting, the most relevant previous values may be any or all of the previous n observations. In Gluon-TS the Transformer is implemented under the name TransformerEstimator [4].

2.8.2 Datasets

Gluon-TS offers 17 datasets ready to be used for training and evaluation, in Table 3 an overview of these are presented. Of the available datasets 7 of them were first used in the M3, M4 and the M5 competitions [38, 37, 39]. The remaining datasets has been used in multiple research papers with various amounts of preprocessing applied to them [44, 33, 49, 70, 69, 24].

Name	Freq	Description
Electricity	Hourly	Hourly electricity consumption of 370 clients sampled between 2011 - 2014. The original dataset on UCL was sampled each 15 minutes whilst the dataset in Gluon-TS had the data resampled into hourly series [4, 52].
Exchange Rate	B	Daily currency exchange rates of eight countries; Australia, Britain, Canada, China, Japan, New Zealand, Singapore and Switzerland between 1990 and 2016 [33].
Traffic	H	This dataset contains the hourly occupancy rate of 963 car lanes of the San Francisco Bay are freeways [3].
Solar Energy	10 Min	The solar power production records in the year of 2006, sampled every 10 minutes from 137 solar energy plants in Alabama [33].
Electricity NIPS	H	The Electricity dataset with additional processing [52].
Exchange Rate NIPS	B	The Exchange Rate dataset with additional processing [52].
Solar Energy NIPS	H	The Solar Energy dataset with additional processing [52].
Traffic NIPS	H	The Traffic dataset with additional processing [52].
Wiki Rolling NIPS	D	The Wiki dataset contains the amount of daily views for 2000 pages on Wikipedia [52].
Taxi	30 Min	Number of taxi rides taken on 1214 locations in New York city every 30 minutes in the month of January 2015. The test set is sampled on January 2016 [52].
M4 Hourly	H	Hourly timeseries used in the M4 competition randomly sampled from the ForeDeCk database [38].
M4 Daily	D	Daily timeseries used in the M4 competition randomly sampled from the ForeDeCk database [38].
M4 Weekly	W	Weekly timeseries used in the M4 competition randomly sampled from the ForeDeCk database [38].
M4 Monthly	M	Monthly timeseries used in the M4 competition randomly sampled from the ForeDeCk database [38].
M4 Quarterly	3M	Quarterly timeseries used in the M4 competition randomly sampled from the ForeDeCk database [38].
M4 Yearly	Y	Yearly timeseries used in the M4 competition randomly sampled from the ForeDeCk database [38].
M5 Dataset	D	Daily Walmart sales for 3049 products across 10 stores [3, 39].

Table 3: Datasets available in Gluon-TS.

2.9 Runtool

As part of an internship I had at Amazon Web Services (AWS) a Python toolkit for creating and executing large scale machine learning experiments was created. This tool, named *the runtool* is open sourced in the Gluon-TS sister repository gluon-ts-tools [19]. Originally the runtool was meant to be a core part of this thesis, however the runtool diverged from the thesis and is now represented here as a third party package.

The runtool works in three parts, first assets such as algorithms and datasets are defined in a YAML config file. This file is then loaded by the runtool from within a python script. Thereafter experiments can be generated via mathematical operators such as + and *. Addition groups algorithms or datasets together into sets and multiplication generates an experiment of the cartesian product of the two sets being multiplied. Finally the runtool starts the generated experiments as training jobs in SageMaker. SageMaker is a cloud service enabling machine learning workloads such as model training and inference to be run on dedicated hardware [57].

Executing a few training jobs is not especially hard via a simple Python script as there are many powerful libraries available such as Gluon-TS. However scheduling and dispatching hundreds or thousands of different training jobs in parallel can quickly become complex. This is further complicated when multiple algorithms with different hyperparameter configurations should be evaluated on multiple different datasets. The runtool simplifies these things via so called $ operators within the config file. These operators enable, for example, inheritance between algorithms or datasets and dynamic updates of values in the config based on what the current experiment is.

In order to execute training jobs on SageMaker via the runtool there are certain requirements:

- The ML model needs to be in a SageMaker compliant Docker container[56].
- The dataset has to be in an AWS S3 bucket [59].
- An IAM role granting the runtool access to the dataset and the Docker image [55].

More information about IAM roles, S3 buckets and how to build Sagemaker compliant docker images can be found in their respective documentation [56, 59, 55] and on the SageMaker homepage [57].

A key benefit of the runtool is its use of config files. Through these, it is possible to rerun any experiment with the exact same configuration as long as one has access to the Docker container, the dataset and the config file. Due to that the

dependencies are built into the Docker image any experiment is fully rerunnable with minimum configuration.

The part of the Runtool which is responsible for starting training jobs is called the jobs dispatcher. While the builtin job dispatcher starts training jobs on SageMaker the runtool is built to make it easy to implement multiple backends. Thus it is possible to extend the runtool such that training jobs can be executed on a local machine instead of on SageMaker.

In Figure 15 a simple config file is displayed which defines two different algorithms and two datasets. This config is then used in the script presented in Figure 1 to create four experiments where both algorithm are executed on both datasets.

```

import boto3
import runtool

# load config file
config = runtool.load_config("config.yml")

# create an experiment
my_experiment = (
  config.myalgo + config.anotherAlgo
  ) * (config.electricity + config.traffic)

# initialize runtool
tool = runtool.Client(
  role="arn:aws:iam::012345678901:role/my_role",
  bucket="my_bucket",
  session=boto3.Session(),
)

# dispatch the jobs
tool.run(my_experiment)  # blocking call


```

Code Fragment 1: Python script using the config from Figure 15 to create four experiments

```
1
2  electricity:
3      meta:
4          freq: 1H
5          prediction_length: 24
6      path:
7          test: file:///path/to/dataset1/train.json
8          train: file:///path/to/dataset1/test.json
9  traffic:
10     meta:
11         freq: 1H
12         prediction_length: 24
13     path:
14         test: file:///path/to/dataset2/train.json
15         train: file:///path/to/dataset2/test.json
16
17 base_algo:
18     hyperparameters:
19         epochs: 10
20     instance: local
21     metrics:
22         MASE: 'MASE\): (\d+\.\d+)'
23         abs_error: 'abs_error\): (\d+\.\d+)'
24 algo1:
25     $from: base_algo
26     image: image_with_algo1
27
28 algo2:
29     $from: base_algo
30     image: image_with_algo2
```

Figure 15: Config file describing two algorithms and two datasets. $from inherits the values from another node.

2.10 Hypothesis Testing

Distributions occur in many areas within nature and science and being able to validate whether different samples come from the same underlying distribution is a long studied problem. This is known as hypothesis testing and several methods have been developed and one can group these tests into two major families, parametric and non-parametric tests [32]. Parametric tests are suitable whenever the type of the underlying distribution is known, otherwise non-parametric tests are more suitable.

The Student's T-test is a commonly used hypothesis test which compares the mean values of two distributions and only works if the samples are independent, normal and they have equal variance [32]. The requirement that the samples should have equal variance can make the T-test unsuitable in practice. However, an alternative to the Student's T-test is the Welch's T-test which allows for different variance between the two samples being tested. The Welch's test has however been shown to be unreliable if the two samples being tested have a considerable difference in size and variance [51].

A well known non-parametric test is the Diebold-Mariano (DM) two sample test which was designed to determine whether two samples of forecasts were statistically different [14]. Hassani et al. showed that the DM test is outperformed by another non-parametric test, the Komogorov-Smirnov two sample test [25].

The Kolmogorov-Smirnov (KS) two sample test is a distance test based on the empirical cumulative distribution function (CDF) of the samples [40]. In essence, the KS test compares the distance between the two CDF, the larger the distance between the CDFs of the two samples, the less likely they are sampled from the same distribution.

3 Approach

The purpose of this thesis is to perform an empirical comparison of forecasting methods. To this end, suitable methods, assets and systems need to be identified and evaluated. This chapter identifies the requirements these need to fulfill in order to be suitable for use in a benchmarking system. Further this chapter identifies assets and methods which need to be investigated further in a experimental study to determine their suitability. In chapter 4 the results of this experimental study is presented. The concepts discussed in this chapter are then combined into a benchmarking toolkit for forecasting methods which is presented in chapter 5. This system is then used for the empirical comparison in chapter 7.

3.1 Defining Reproducibility

In Section 2.7 the term reproducibility was split into three categories; *technical*, *statistical* and *conceptual*. A benchmarking toolkit should adhere to the requirements posed by these rules in order to be reproducible.

In addition to the requirements identified in Section 2.7, automating the benchmark would lead to a higher degree of reproducibility as the benchmarks would be standardized and thus less error prone. However, certain aspects of a benchmark, such as the error metric, are not suitable to be standardized since suitable error metrics depends on the use case.

A summary of the requirements identified in this section for enabling reproducible benchmarks is presented in 16.

3.2 Defining Fair Comparisons

The adjective Fair is defined as *acceptable and appropriate in a particular situation.* in the Oxford Dictionary. When benchmarking forecasting models, one method of achieving fair comparisons is to evaluate algorithms on multiple complementary datasets as this showcases the strengths and weaknesses of algorithms in different scenarios. Additionally, fair comparisons also require that the variance is shown as this provides more detail of an algorithms overall performance than a single value [11].

1. Encapsulation tools should be used for dependency and algorithm management.

2. Experiments should run multiple times in order to build up a distribution of errors such that the variance of the error is recorded.

3. Statistical tests should be applied to the results to enforce statistical reproducibility.

4. Multiple datasets should be used when benchmarking to cover various use-cases for conceptual reproducibility.

5. The benchmark should be automated

Figure 16: Requirements for a reproducible benchmark.

Simply executing an algorithm on multiple datasets is not enough as it is important to give each algorithm a fair opportunity to perform well on each dataset. This is normally achieved by *tuning* the hyperparameters of an algorithm for the dataset in question. A fair way of tuning algorithms is however non trivial as different strategies tend to be biased [61]. One method of achieving fairness could be to limit the time spent for hyperparameter tuning to some constant value. This would however be unfair towards slower methods such as DNNs or RNNs as the search space traversed would be smaller when compared to faster models such as simple NNs or classical models. Another approach is to limit the size of the search space by limiting the number of hyperparameter configurations to be tested. This would benefit models with few hyperparameters as each hyperparameter would be more thoroughly tuned than if more hyperparameters would be available. The inverse of this would also hold, one could limit the number of configurations for each hyperparameter tested but let the total number of hyperparameter configurations to be unbound. This would however be unfair towards models with few parameters.

Another approach would be to tune each algorithm such that it achieves its *optimal* performance for each dataset. This would be fair in the sense that it is independent of the complexity of the algorithm being tested. For ML competitions this is commonly the case as contestants tune their algorithms to perform optimally in order to win [50]. This is also the case when algorithms are used in practice as companies and organizations leveraging machine learning, spend considerable effort in tuning these to fit their data [9].

To enable fair comparisons the implementations of the models being compared should be bug free and optimized. This is especially important for very com-

plex models as the engineering effort of implementing such algorithms correctly is considerable.

A summary of the requirements for fair comparisons which have been identified is presented in 17

1. Optimally tuned algorithms for each dataset
2. Variance should be reported
3. Models should be evaluated on multiple complementary datasets
4. Reference implementations of tested forecasting models should be used when possible

Figure 17: Requirements of a Fair Benchmark.

3.3 Defining Accurate Comparisons

In Section 2.4.1 different metrics were presented for comparing forecasting accuracy. It was established that error metrics have different strengths and weaknesses which makes different metrics suitable for different domains or applications. Thus, in order to perform an accurate comparison of forecasting methods via a benchmarking system, multiple different metrics need to be made available.

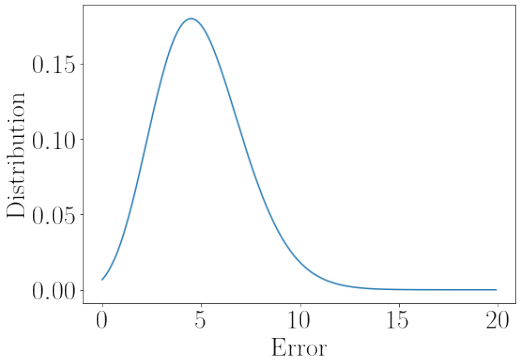

Figure 18: Poisson distribution

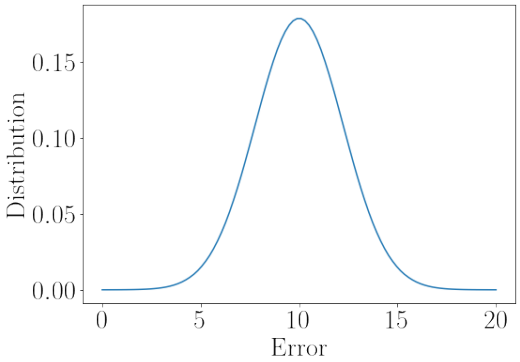

Figure 19: Gaussian distribution

Figure 20: Which error distribution is better?

Since, for reproducibility purposes, each model shall generate a distribution of error metrics one cannot any longer compare algorithms based on single metric

values. In Figure 20 two examples of possible error distributions are presented. Since a smaller error is generally considered better, it makes sense that distributions which are concentrated around zero perform the best. For example in Figure 20 an algorithm generating an error distribution such as the poisson is more likely to produce errors close to 0 than an algorithm with an error distribution like the normal distribution in Figure 19.

A summary of what is required to perform accurate comparisons is presented in 21.

1. Suitable error metrics should be used for the domain and datasets.
2. The distribution of the error metric over multiple runs is used when comparing different algorithms.

Figure 21: Requirements for accurate comparisons

4 Designing a Benchmarking System

A benchmarking system for benchmarking of forecasting methods should fullfill the requirements defined in Chapter 3. These requirements, summarized in Table 22, require further specification. For example; which encapsulation system should be used, what is a suitable hypothesis test for statistical reproducibility, how should distributions of errors be compared and how should a representable subset of datasets be chosen. This chapter answers these questions and presents a benchmarking architecture which enables fair, accurate and reproducible comparisons of forecasting algorithms.

Requirement	Accurate	Fair	Technically reproducible	Statistically reproducible	Conceptually reproducible
Multiple error metrics available	X				
Compare distributions of error metrics	X	X	X	X	X
Optimally tuned models		X	X		
Multiple datasets used		X			X
Public datasets used			X		X
Reference code for ML models		X	X		
Use encapsulation tools		X	X		
Use statistical tests				X	

Figure 22: Requirements of a fair, accurate and reproducible benchmarks.

4.1 Design Considerations

This section investigates how different tools and techniques can be combined in order to create a benchmarking system for fair, accurate and reproducible comparisons of foreasting methods as per the requirements presented in Table 22.

Due to the popularity of Docker and its native capability to encapsulate code and dependencies, Docker containers are chosen as the encapsulation tool for this project. An advantage with docker containers is that popular tools such as Github, AWS, Azure and Dockerhub allows for storage and sharing of Docker images which makes it trivial for researchers to distribute their algorithms in a ready to use package.

Docker containers are however not sufficient for complete technical reproducibility as arguments such as hyperparameters or the number of CPU cores should be used can be passed to the containers when training them. If all arguments are not supplied when presenting these algorithms, technical reproducibility becomes hard to achieve. The Runtool described in Section 2.9 simplifies these things as the config file used when running the experiment can be used by a third party to rerun exactly the same experiment as long as the Docker image and dataset is available.

Reference implementations of complex algorithms are required for fair comparisons, Gluon-TS which was presented in Section 2.8 offers several algorithms ranging from simple naive solutions to highly complex DNN hybrid models. Furthermore, Gluon-TS offers ready to use Dockerfiles for both CPU and GPU execution of these algorithms. An additional benefit of leveraging Gluon-TS is that multiple datasets are available for training and testing algorithms on and that multiple error metrics are calculated as part of their backtesting system.

As per the discussion in Section 3.2 a fair and conceptually reproducible comparison should make use of multiple datasets with different complementary characteristics. Gluon-TS offers several datasets used in research papers and competitions, and an analysis of these needs to be performed to identify a representable subset. Such an analysis is performed in Section 4.5

In order to capture distributions of error metrics, the seed of the encapsulated algorithms should not be set from within the Docker image otherwise rule 2 of Figure 16 would be violated as each run would be deterministic.

Since two sample hypothesis tests can be used for the purpose of verifying wether two distributions of error metrics are sampled from the same underlying distribution this type of statistical test is chosen to enforce rule 3 of Figure 16. Three hypothesis tests for comparing distributions for reproducibility purposes were discussed in Section 2.10. However the practical performance of such tests when applied to real world distributions of error metrics need to be examined. Particularly, the amount of data needed for them to be accurate needs to be determined so

that suitably large distributions of error metrics are collected when benchmarking. In Section 4.6 such an analysis is performed.

It is common for benchmarks and competitions to generate tables where different forecasting models are compared using real valued numbers to describe their performance. Normally this number is an error metric [37, 38, 39, 28, 53, 44]. Similarly, it would be suitable if a real number could be used to summarize the distribution of error metrics. The conversion from a distribution to a single number is unlikely to be lossless, however it is important that the conversion should accurately represent the distribution by benefiting errors close to 0 and punishing errors further away. Such a conversion should not be limited by the error metric used in the distribution but be applicable to any error metric otherwise the *"Multiple error metrics aviailable"* requirement in Figure 22 would be violated.

Tuning of algorithms is a requirement for fair comparisons, thus the possibility of hyperparameter tuning should also be available as part of the benchmarking suite. Since many ML models exhibit non-deterministic behaviour, each hyperparameter configuration being tested should be executed multiple times and aggregated, otherwise rule 2 from Figure 16 would be violated.

To summarize, Docker images are chosen to be the encapsulation tool. Further, the Runtool will be used to execute this images as it makes backtests technically reproducible per design. Furthermore, the datasets and forecasting algorithms available in GluonTS will be used for the benchmark as these are publicly available and the models are well tested reference implementations. Hyperparameter optimization should be applied to algorithms to enable fair comparisons, and for models with non-deterministic output, each configuration should be executed multiple times when tuning. Benchmarks shall be executed multiple times to collect a distribution of error metrics and a suitable aggregation method for ranking error distributions needed. In order for benchmarks to be statistically reproducible two error distributions from the same algorithm on the same dataset and with the same hyperparameters should pass a two sample hypothesis test.

4.2 Proposed Benchmarking System

This section introduces a benchmarking system based on the design consideration in Section 4.1. In Figure 23 an overview of the proposed benchmarking system is displayed. Here the benchmark starts by a user providing the runtool configuration file containing the algorithm to use, the name of the image to use and other data such as which hyperparameters to pass to the algorithm. The system then downloads the datasets identified in 4.5 to the local machine for later use.

The benchmarking loop is as follows: the algorithm configuration file is first loaded into the runtool and an experiment is generated where the algorithm is

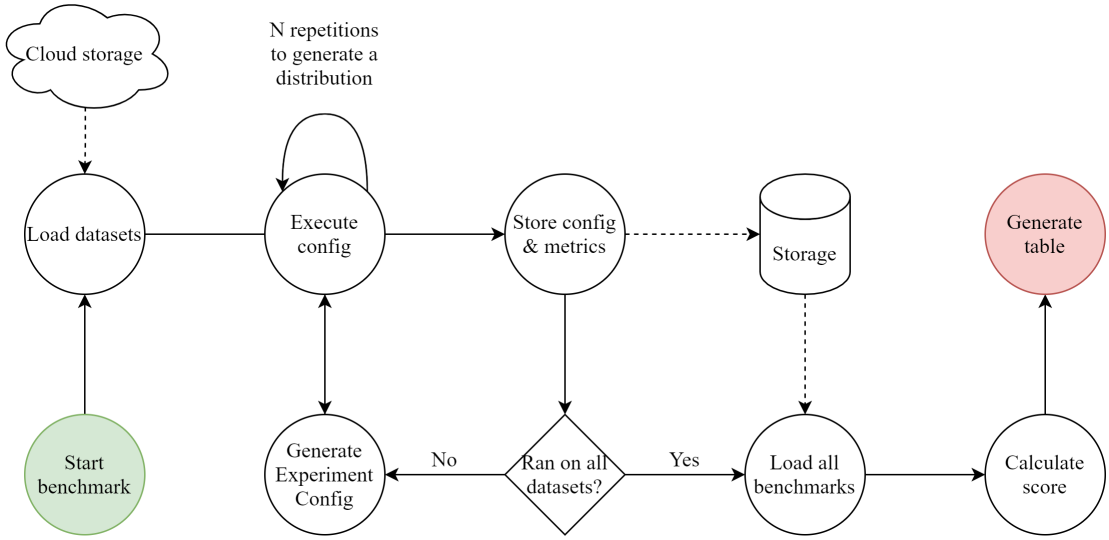

Figure 23: Proposed benchmarking system.

executed on each of the datasets N times. The value of N is determined in Section 4.6. Each time the algorithm is executed, error metrics are reported by the GluonTS Backtesting functionality and these logs are captured by the Runtool and stored to a file or database for long term storage. This file will be referred to as the *Benchmark Result File* (BRF) for the remainder of the thesis. Along with the distribution of error metrics, the config file used should be stored in the BRF to make it possible to rerun each benchmark.

The latest benchmarks error distribution is then aggregated through some scoring method which summarizes the distribution as a single value. This is then repeated for each previous benchmark which has been run. The resulting scores are then used to rank the algorithms against each other. A table is then presented to the user showing the relative rank of the latest benchmark compared to the previous benchmarks.

4.3 Reproducing Benchmarks

In order to both technically and statistically be able to reproduce a benchmark a user would need access to the image used and the BRF created by the benchmarking system. Provided these, the verification system could rerun the benchmark using the config file stored in the BRF and the provided image. Rerunning a benchmark with the same setup should result in a similar distribution being generated by the algorithm. Statistically speaking, if the algorithm in the image would

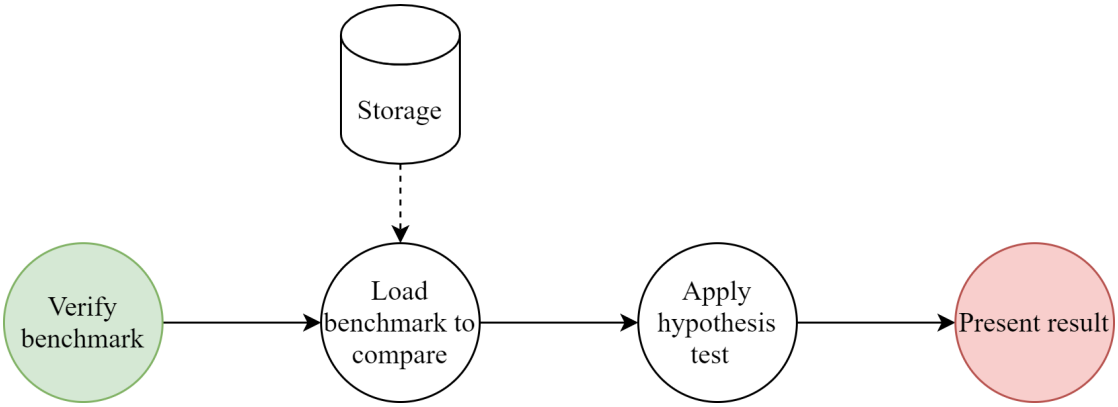

Figure 24: Proposed system for verifying benchmarks.

be the same as the reference algorithm, both error metric distributions would be sampled from the same underlying distribution. The new distribution can then be compared with that of the benchmark stored in the BRF through a suitable hypothesis test.

4.4 Hyperparameter Tuning

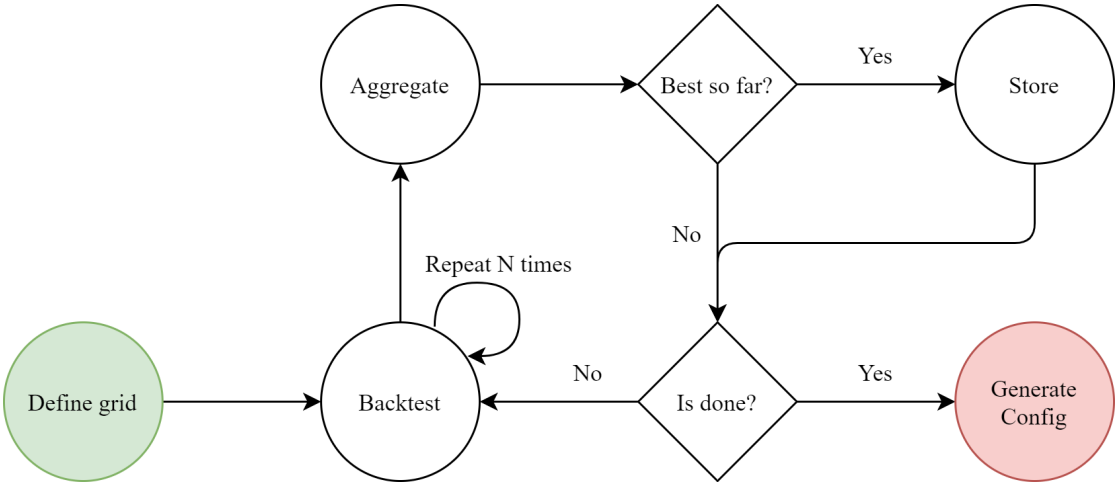

Figure 25: Proposed system for hyperparameter optimization with repeated runs.

Hyperparameter tuning is a non-trivial task in itself and many advanced solutions exist such as Bayesian hyperparameter search, hyperband search or advanced meta learning approaches [63, 15, 34]. The hyperparameter tuning approach pro-

posed here is not intended to supersede these. Instead it focuses on taking the non deterministic output of each run into account when tuning.

The proposed algorithm is in essence a simple grid search tuning algorithm with one major difference. Each configuration in the tuning loop is executed multiple times and then aggregated before being evaluated. Common ML frameworks which offer grid search such as SageMaker or SciKit-Learn (sklearn) surprisingly lack the capability to backtest each configuration multiple times [57, 45]. While sklearn does offer a cross validation version of its grid search implementation, *GridSearchCV*, this splits the dataset into multiple parts for each iteration. Thus it does not serve the same purpose as performing the backtest multiple times and aggregating the results.

Grid search was chosen as the tuning approach for this system as it is conceptually simple and it is a well known method. It does however have the downside that many unfavorable hyperparameter configurations are evaluated, something which e.g. bayesian search avoids [63].

An issue with the suggested tuning architecture is that time for tuning increases linearly with the number of repeated runs. Due to this, it may be suitable in real life scenarios to apply this approach as a second step after a set of promising hyperparameter configurations has been identified by another more time efficient approach.

4.5 Dataset Analysis

It is common that new forecasting methods are compared on several datasets as to showcase their predictive power in different scenarios. Different datasets exhibit different characteristics such as trend and seasonality. By identifying a representable set of datasets with complementary characteristics, one can evaluate how robust a forecasting algorithm is to different datasets. Choosing a representable subset of datasets is also needed as the time to train an algorithm scales with the amount of datasets that it should be trained on. For an empirical comparison such as this, it is unfeasible to run training and tuning jobs for all algorithms and datasets available in Gluon-TS since this would take too long. In the remainder of this section an analysis of the datasets available in Gluon-TS is performed to identify a representable subset with diverse characteristics.

4.5.1 Methodology

For each dataset in Gluon-TS plots of the average time series along with one standard deviation from it is generated. This is done to get an overview of what the datasets looks like. Plotting timeseries for this purpose is common practice in

time series forecasting, or as Hyndaman et al. so eloquently put in Forecasting: Principles and practice: *"The first thing to do in any data analysis task is to plot the data."* [28].

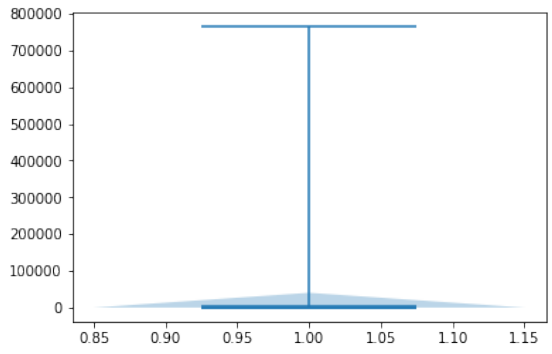

Figure 26: Unscaled violin plot

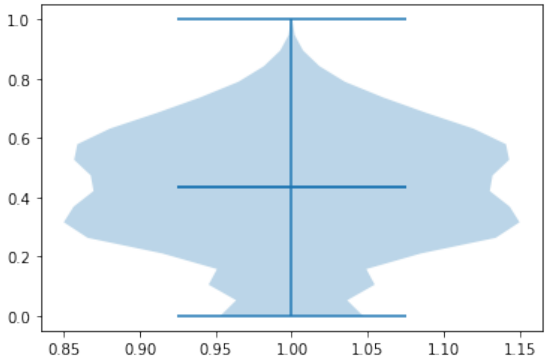

Figure 27: Scaled violin plot

Figure 28: Violin plots of the Electricity dataset with and without scaling.

A common tool for visualising timeseries is to generate a histogram of the time-series data. However as the datasets available in Gluon-TS contains tens of thousands of timeseries, plotting each of these individually is unfeasible. Instead the average timeseries and one standard deviation of its values is plotted. Another common method for visualizing single timeseries is to generate a histogram of its values [28]. Instead of generating such histograms violin plots are used in as these capture both the distribution of values just as histograms do as well as the median value and the interquartile range. Since each timeseries is on its own scale, the aggregate violin plots are hard to read. By scaling each timeseries in the dataset by the maximum value of that timeseries before plotting it the violin plot becomes more easy to interpret.

In addition to the visual tools the following statistics is calculated for each of the datasets in Gluon-TS:

- Mean value
- Max value
- Min value
- Number of timeseries
- Number of datapoints
- Length of shortest timeseries

- Length of longest timeseries
- Strength of the trend (mean and standard deviation)
- Strength of the seasonality (mean and standard deviation)

Simple metrics such as the maximum, mean and minimum values for are useful to get a high level overview of the dataset and to identify possible issues which can surface further down the line. For example, the MAPE metric is unstable for timeseries close to zero as a division with zero occurs [28].

The number of timeseries are important for global models as these then have more data which they can learn across which help to combat overfitting. However, the number of timeseries is not important for local models as these are only impacted by the quality and length of each individual time series. Longer timeseries however, benefit forecasting models which can handle longer horizons [38].

For these reasons, the number of timeseries and the lengths of the shortest and longest timeseries are necessary metrics when comparing datasets. Optimally one would have both long timeseries and many series as this would benefit both global and local models.

K-fold cross validation as discussed in Section 2.4 is a method for increasing the number of series in a dataset. Most of the datasets in Gluon-TS have had K-fold cross validation applied to them already. Thus the potential size after applying K-fold cross validation should be considered for those datasets for which it has not yet been applied.

In chapter 2 it was shown that the strength of seasonality (SoS) and the strength of the trend (SoT) of timeseries are useful metrics for differentiating timeseries. Due to this, these two measures are the main criterias by which the datasets are chosen. Since each dataset can have both a SoT and SoS, the most diverse datasets are those with complimentary SoT and SoS. In total four datasets can then be identified which fullfill these criteria; one with both high SoT and high SoS, one with high SoT and low SoS, one with low SoT and high SoS and one with both low SoT and low SoS. Since each time series within a dataset is unlikely to have the same SoT or SoS also the variance of the SoT and SoS need to be taken into account. Thus, when two datasets have similar strength, the one with the lowest variance is chosen. This emphasizes the diversity of the datasets as two datasets with high variance in SoS or SoT are prone to contain more similar timeseries having similar SoT and SoS.

Since larger datasets make models less prone to overfit, the largest dataset is chosen in case of a tie where two datasets are exhibiting a negligible difference in SoT and SoA. A summary of the criterias on which the datasets are evaluated is shown in Table 29.

1. One dataset with high trend and low seasonality
2. One dataset with low trend and high seasonality
3. One dataset with low trend and low seasonality
4. One dataset with high trend and high seasonality
5. Lower variance of strengths is prefered
6. Larger dataset chosen in case of a tie

Figure 29: Criteria for identifying a representable subset of datasets.

Limitations

The M4 dataset does not significantly differ to the M3 dataset except for its size [64]. Thus, the M3 dataset is not considered in this comparison. Another set of datasets which are not going to be used are the NIPS datasets. This is due to them having had postprocessing applied to them. As the exact post processing is not known, comparing any findings when training on these datasets with other papers becomes hard and introduces uncertainty.

4.5.2 Result

To calculate the strengths of the trend and seasonality a STL decomposition is needed for each timeseries in the dataset. This decomposition is done using the STL method of the Statsmodel package in Python [54]. The strengths of each timeseries is then calculated using the formulas in Chapter 2. In order to summarize the strengths, the average and the standard deviation is then calculated for each dataset.

The complete plots and statistics are available in the appendix as they were to numerous to be displayed here. However a summary of the extracted statistics is presented in Table 4.

There is only one dataset in Table 30 which exhibits low trend and low seasonality and that is the M5 dataset. However the M5 dataset does show the most variance of all the datasets which implies that it contains many timeseries with and without trend and seasonality. Despite this, since it is the largest dataset available and that there are no other datasets which exhibit low average trend & seasonality, the M5 dataset is deemed appropriate for use in the benchmarking system.

Dataset	Trend	Seasonality	Trend Dev.	Seasonality Dev.
Exchange Rate	1.00	0.12	0.00	0.30
M4 Daily	0.98	0.05	0.05	0.10
M4 Yearly	0.93	0.09	0.13	0.16
M4 Quarterly	0.90	0.20	0.16	0.27
M4 Monthly	0.84	0.32	0.32	0.30
M4 Weekly	0.77	0.31	0.31	0.35
Electricity	0.65	0.84	0.17	0.19
M4 Hourly	0.62	0.88	0.37	0.16
Wiki Rolling	0.53	0.23	0.27	0.26
M5	0.38	0.28	0.32	0.33
Traffic	0.16	0.67	0.12	0.10
Solar Energy	0.09	0.84	0.03	0.02
Taxi	0.02	0.66	0.02	0.08

Figure 30: Strength of trend & seasonality for datasets in Gluon-TS.

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Freq.
Elec.	2510.68	321	6755124	21044	21044	0.0	764000.0	1H
Elec.	2509.92	2247	47501580	21068	21212	0.0	764000.0	1H
Exch.	0.68	8	48568	6071	6071	0.01	2.11	1B
Exch.	0.68	40	246440	6101	6221	0.01	2.11	1B
Solar	40.35	137	960233	7009	7009	0.0	509.05	10min
Solar	40.25	959	6813695	7033	7177	0.0	509.05	10min
Traf.	0.06	862	12099032	14036	14036	0.0	0.72	H
Traf.	0.06	6034	85272488	14060	14204	0.0	0.72	H
Exch.*	0.68	8	48568	6071	6071	0.01	2.11	B
Exch.*	0.68	40	246440	6101	6221	0.01	2.11	B
Elec.*	607.95	370	2142282	1081	5833	0.0	168100.0	H
Elec.*	652.36	2590	10340239	1105	4000	0.0	168100.0	H
Solar*	40.35	137	960233	7009	7009	0.00	509.05	H
Solar*	40.25	959	6813695	7033	7177	0.00	509.05	H
Traf.*	0.05	963	3852963	4001	4001	0.00	1.00	H
Traf.*	0.05	6741	26964000	4000	4000	0.00	1.00	H
Wiki	3720.54	9535	7551720	792	792	0.00	7752515.00	D
Wiki	3663.55	47675	40619100	792	912	0.00	7752515.00	D
Taxi	8.79	1214	1806432	1488	1488	0.0	265.0	30min
Taxi	7.41	67984	54999056	149	1469	0.0	225.0	30min
M4 H	6827.69	414	353500	700	960	10.00	703008.00	H
M4 H	6859.56	414	373372	748	1008	10.00	703008.00	H
M4 D	4951.40	4227	9964658	93	9919	15.00	352000.00	D
M4 D	4960.15	4227	10023836	107	9933	15.00	352000.00	D
M4 W	3738.52	359	366912	80	2597	104.69	51410.00	W
M4 W	3755.97	359	371579	93	2610	104.69	51410.00	W
M4 M	4193.28	48000	10382411	42	2794	20.00	132731.31	M
M4 M	4207.51	48000	11246411	60	2812	20.00	177950.00	M
M4 Q	4141.00	24000	2214108	16	866	19.50	82210.70	3M
M4 Q	4287.13	24000	2406108	24	874	19.50	82210.70	3M
M4 Y	3630.52	23000	715065	13	300	22.10	115642.00	12M
M4 Y	4076.24	23000	852909	19	300	22.00	158430.00	12M
M5	1.12	30490	57473650	1885	1885	0.00	763.00	D
M5	1.13	30490	58327370	1913	1913	0.00	763.00	D

Table 4: Statistics of the datasets, top row for each dataset is the train split, the bottom row is the test split.

In order to choose a dataset which fits into the category of high seasonality and low trend there are three options; Solar Energy, Taxi and Traffic. Of these the Solar Energy dataset has the lowest variance for both the trend and the seasonality. This in addition to having the second lowest average trend of the three and $\tilde{2}3$ % higher average seasonality than the Taxi and the Traffic datasets makes it most suitable according to the criteria in Figure 29.

The datasets with the lowest seasonality and the highest trends are the M4 datasets except for the M4 Hourly as well as the Exchange rate dataset. The one with the lowest variance and the highest trend of these is the Exchange Rate dataset closely followed by the M4 Daily dataset. Since the M4 Daily has lower variance for both strength and seasonality, and since the size of the Exchange Rate dataset is much smaller than the M4 Daily dataset, with only 8 timeseries and 48k datapoints in comparison to the 4227 series and 9.9M datapoints the M4 Daily dataset is more suited for use in the benchmarking system. Furthermore, all other M4 datasets exhibit a higher variance than the M4 Daily which enforces the M4 Daily to a better choice for this comparison.

When it comes to finding a dataset which exhibit both high trend and a high seasonality, there are only two options, the Electricity dataset and the M4 Hourly dataset. They both have a high variance, however the variance of the Electricity dataset is slightly lower than that of M4 Hourly. In addition, the Electricity dataset has almost twice the amount of datapoints as M4 Hourly. Thus the Electricity dataset is chosen as it fits the criteria better.

To summarize the four datasets which most closely fit the criteria defined in 29 are:

	Low trend	**High Trend**
Low seasonality	M5	M4 Daily
High Seasonality	Solar Energy	Electricity

Table 5: Datasets with complimentary strengths of trend and seasonality

4.6 Comparing Tests for Validating Forecasting Performance

This section investigates the performance of three hypothesis tests for use on distributions of error metrics. The investigated tests are the T-test, Welch's T-Test and the Kolmogorov-Smirnov two sample test.

Gluon-TS catches accuracy regressions through checking whether new forecasts are within 1.645 standard deviations, i.e. within the 95th percentile of past values

- Which test is best for limited data
- Is a parametric test suitable or is a non-parametric test needed
- What is a reasonable sample size
- Which test has the lowest false reject ratio for related distributions
- Is a naive standard deviation based test sufficient

Figure 31: Questions for identifying a suitable hypothesis test.

[3]. This approach is based on the assumption that the distribution of error metrics is normally distributed. If this is True then, this simpler approach may be more suitable than a hypothesis test for reproducibility. Thus this approach will be evaluated along with the hypothesis tests.

In order to decide which tests are suitable certain aspects of the tests and the data need to be investigated. Since the naive test and the parametric tests rely on that the distributions are well defined to function it is important to identify how error distributions of forecasting methods look like, i.e. is it normally distributed or do they follow other distributions.

Different tests require different amount of samples to be accurate, and some tests can become unstable if the sample sizes differ too much [25, 51]. Since performing a benchmark is time consuming, it is beneficial from a time perspective to collect as few samples as possible. Thus identifying how the tests perform when applied to different sample sizes is important as a minimum sample size can be identified.

A summary of the questions this section aims to answer is presented in Figure 31.

4.6.1 Methodology

Several of the questions in Figure 31 require distributions of error metrics to be collected. For this purpose, the DeepAREstimator presented in Section 2.8.1 is suitable as it is both quick to train and a non-trivial forecaster. The dataset chosen is the Electricity dataset since it is a medium to small sized dataset, thus shortening the required training time.

The Runtool is then used to create 900 experiments where the DeepAREstimator is executed on the Electricity dataset. 300 of these runs has the hyperparameter *distr_output* set to StudentTOutput, 300 runs are executed with *distr_output* set to use the NegativeBinomialDistribution. The remaining 300 runs are executed using the Poisson distribution. The three distributions generated by these 900

training jobs are referred to as NB, ST and P in this chapter.

After the distributions are collected, histograms of the distributions are created to identify whether the distributions are visually different. This is done since if all three are visually similar it is unlikely that they can be used for evaluating the hypothesis tests. Further, a visual analysis can determine whether the naive method and or the T-test is suitable since it requires normally distributed data.

After the visual analysis, the ratio of false rejects are calculated, i.e. how often the test is unable to detect that two samples are sampled from the same distribution. To answer this questions, let N be the total distribution and X, Y be the two samples drawn from N. The size of X is then defined by

$$k \leq |X| \leq |N| - |Y|$$

where k is a constant value. Similarly, the size of Y varies between:

$$k \leq |Y| \leq |N| - k$$

For each sample of X and Y the tests are applied and false negatives are recorded.

In order to evaluate a suitable minimum sample size, heatmaps are generated where the tests are applied to the three distributions with varying sample sizes. Each test is performed with a *P value* of 0.05. If the test accepts the samples to be from the same distribution the color green is used while a rejection, is colored red.

4.6.2 Results

The histograms shown in Figure 35 do not resemble a normal distribution. Instead all three histograms are more closely approximating a log normal distribution. Note that the histogram in Figure 32 and 33 are very similar while the one in Figure 35 differs. This indicates that different hyperparameter configurations for a single algorithm impacts the error distribution.

In Table 6 the amount of false rejections of the three hypothesis tests and the naive solution is presented. From this data it is clear that the Naive solution is the best performing with a 1% chance of failing to validate that the two samples are from the same distribution. The Kolmogorov-Smirnov performs second best, however it is incorrect 21% of the time when evaluating on the distribution generated by the Poisson configuration. The two T-Tests are the worst performers, performing worse than or equal to the Kolmogorov-Smirnov test on all distributions.

Since the distributions of error metrics are not reminiscent of the normal distribution, the naive method, the T-test and the Welch's T-test which are parametric tests are unsuitable for use when verifying dataset distributions. The data in Table

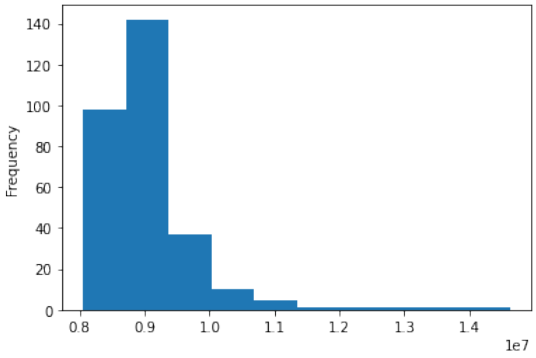

Figure 32: Student-T

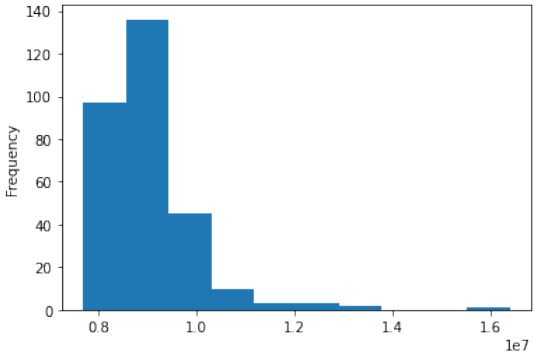

Figure 33: Neg-Binomial

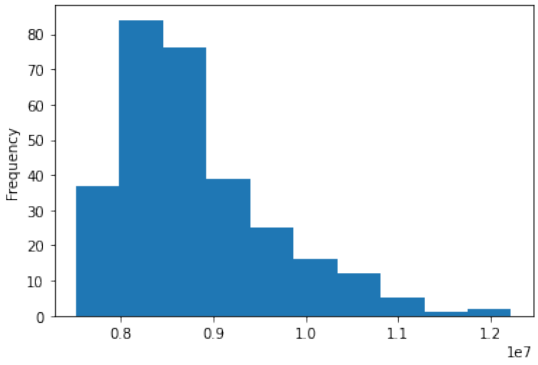

Figure 34: Poisson

Figure 35: Histograms of the absolute error for 300 runs of DeepAR on the Electricity dataset with three different values of the hyperparameter *distr_output*

6 further enforces this since Kolmogorov-Smirnov is the second best performer for all distributions where only the naive method is better. In Figure 38 the naive method is evaluated on the Poisson and Negative Binomial distributions. It is clear from this heatmap that the naive method is prone to accept quite different distributions for all but small sample sizes. This behaviour is also apparent on all other heatmaps where the naive method is used, see Appendix **??**. This is expected since the distributions all have similar average values even though their distributions are visually different. In comparison, the Kolmogorov-Smirnov test is able to differentiate all distributions given sufficient sample sizes, see Figures 36, 37, 39.

The sample size required to differentiate between different distributions (true negatives) as well as the sample sizes required for true positives for the Kolmogorov-

Name	%	**c**onfiguration
Naive	<1%	Negative Binomial
Naive	<1%	Student T
Naive	<1%	Poisson
T-Test	5%	Negative Binomial
T-Test	5%	Student T
T-Test	32%	Poisson
Welchs T-Test	11%	Negative Binomial
Welchs T-Test	8%	Student T
Welchs T-Test	31%	Poisson
Kolmogorv-Smirnov	3%	Negative Binomial
Kolmogorv-Smirnov	5%	Student T
Kolmogorv-Smirnov	21%	Poisson

Table 6: Number of times when the algorithm failed to recognize that the samples were from the same distribution

Smirnov test is presented in Table 7. This table summarizes the heatmaps generated for the Kolmogorov-Smirnov test when applied to samples of varying sizes from the three distributions as described in 4.6.1. Additional heatmaps not shown in this chapter are available in Appendix **??**.

	Student T	Poisson	Negative Binomial
Student T	> 50		
Poisson	> 25	> 45	
Negative Binomial	> 160	> 60	< 130

Table 7: Required sample sizes for the Kolmogorov Smirnov test on three distributions of error metrics.

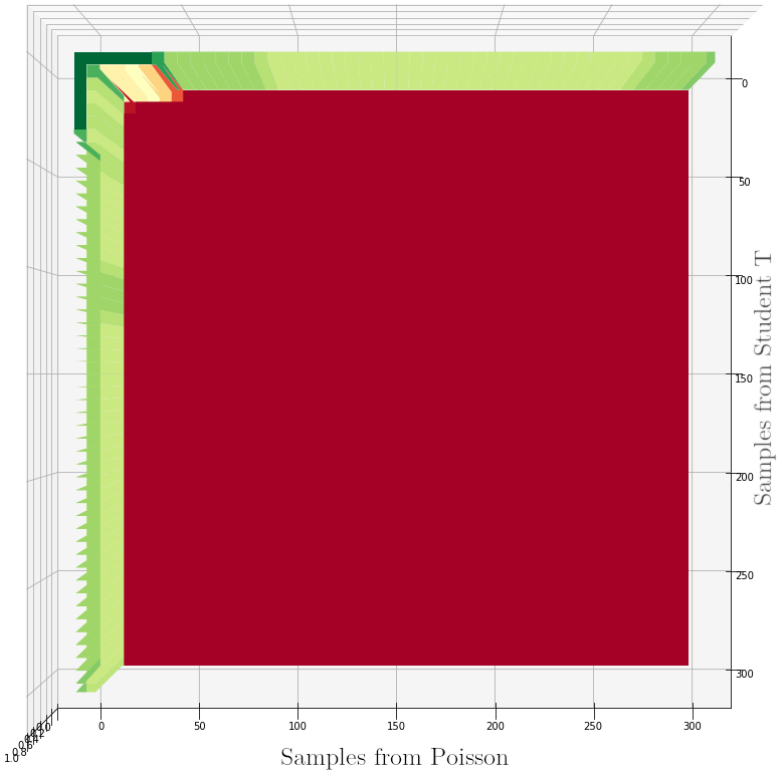

Figure 36: Heatmap of KS applied to the ST and P distributions.

From Table 7 it is shown that the KS test becomes accurate for true positives at a sample size of at least 40 samples. However, it seems as if KS can become unstable for sample sizes above 130. To be able to use the KS statistic to distinguish between different distributions, sample sizes above 160 may be needed if the distributions are visually very similar, otherwise, sample sizes above 60 suffice. These values are aligned with the findings of the simulation study performed by Hassani et al. **??**. There they showed the need for > 128 samples to achieve $> 99\%$ rejection rate for similarly looking distributions.

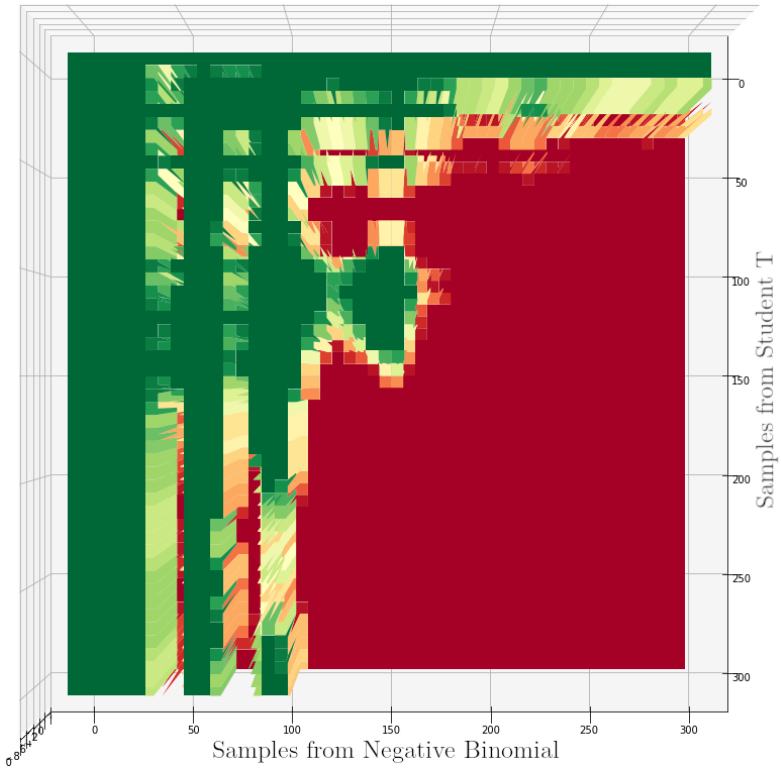

Figure 37: Heatmap of KS applied to the ST and NB distributions.

To summarize, despite the limited data available for this comparison, the Naive test, the T-test and Welch's T-Test has been shown to be unsuitable for comparing distributions of time series forecasting errors due to non-normality of the data. This makes the Kolmogorov-Smirnov two sample test the best suited statistical test for asserting the statistical reproducibility of error metric distributions. When evaluating the Kolmogorov-Smirnov on samples from three different distributions it was shown that > 60 samples are required to verify distributions but more than 160 may be required to differentiate similar distributions.

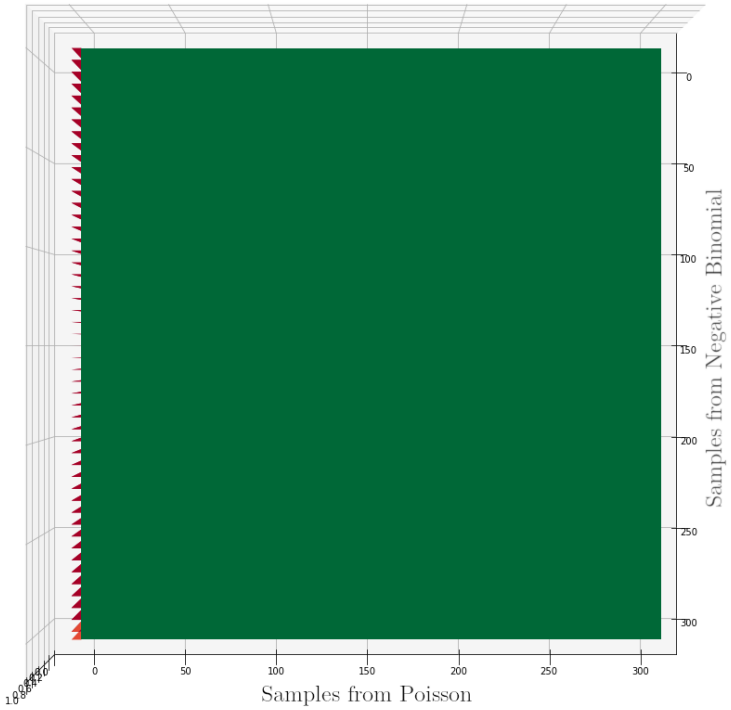

Figure 38: Heatmap of the Naive method applied to the NB and P distributions.

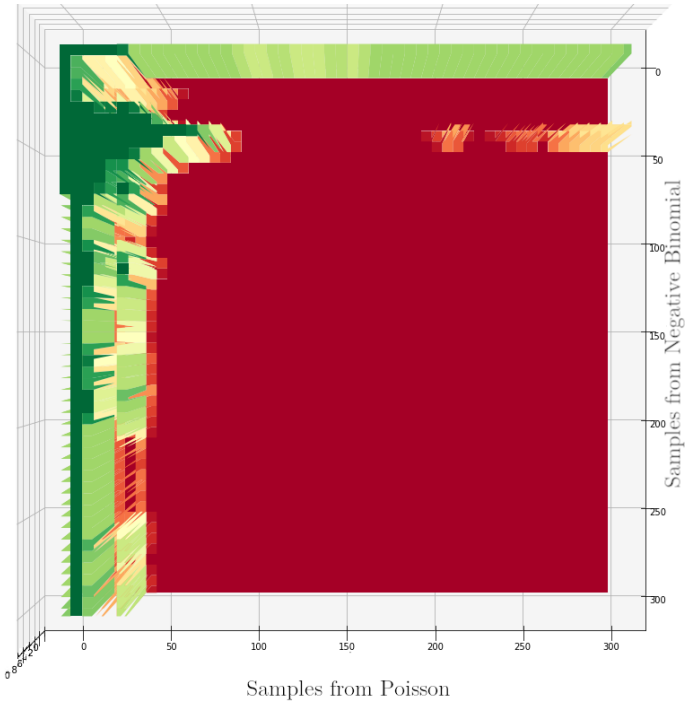

Figure 39: Heatmap of KS applied to the NB and P distributions.

4.7 Ranking Distributions

To rank algorithms based on distributions of error metrics, a suitable metric need to be calculated. Since perfect forecasts would result in an error of zero, algorithms where the distributions are close to 0 should recieve a better score than those further away. In Figure 48 six possible error distributions are shown. Only non-negative distributions are considered here as many common error metrics such as; MAE, RMSE, MSE, MASE, MAPE are non-negative metrics [3, 28]. Furthermore, a distribution with negative values can converted to a non-negative distributions by taking the absolute value of each element in it.

An optimal ranking of the error distributions in Figure 48 would favor those most often close to 0 and punish those with heavy tails away from zero. In addition to this a score should favor consistency, i.e., low variance, thus multimodal distributions with one peak close to zero and one further away should perform worse than distribution with a single peak in between.

A metric which fits these criteria in the realm of time series forecasting is the Root Mean Squared Error which was presented in Section 2.4.1. The equation for RMSE is:

$$RMSE = \sqrt{mean((Y - \hat{Y})^2)}$$

Figure 40: Equation for calculating RMSE

Through fixing the value of the target value Y to 0 and viewing each datapoint in the error distribution as a forecast $\hat{Y}$ this metric can be used to aggregate distributions of error metrics where the optimal value is 0. This modified version of RMSE is referred to as RMSE4D (RMSE for Distributions) for the remainder of the thesis.

$$RMSE4D = \sqrt{mean(\hat{Y}^2)}$$

Figure 41: Equation for calculating RMSE4D

A RMSE4D close to 0 is considered optimal and it is a non-negative error, thus, when comparing two RMSE4D values, lower values are better. An issue with RMSE is that the $\hat{Y}^2$ part of the equation makes it sensitive to outliers in the data. Thus when calculating the RMSE4D the top 5% and the bottom 5% should be removed to account for this.

4.7.1 Evaluation of RMSE4D

To evaluate whether RMSE4D is suitable for aggregating distributions of error metrics, it is evaluated against two simple aggregation techniques, the mean and the median values. These methods are evaluated on six samples of 100 datapoints taken from the distributions shown in Figure 48. These six distributions were designed to be as different as possible to identify potential issues with the three aggregation methods.

Histograms of the samples used for the evaluation are shown in Figure 55. All of these distributions except for the Exponential in Fig. 50 are generated by randomly sampling 100 samples from one or more normal distributions. For brevity a normal distribution with mean μ, and deviation σ is written as $N(\mu, \sigma)$. If two different distributions are combined then 50 samples are taken from each distribution and the combination is expressed as $N(\mu_1, \sigma_1) + N(\mu_2, \sigma_2)$. Furthermore for all distributions in Figure 55 the absolute value has been taken of the 100 random samples of the distributions since only non-negative errors are considered.

An optimal ranking of the 6 distributions would be the Exponential in first place since all errors are distributed close to 0. The N(2,2) distribution should be ranked second since no other distribution are close to 0. In third place the N(5,2) distribution in Figure 42 or the Heavy tailed distribution in Figure 53 fits since consistent performance is preferred, but the peak of the heavy tailed distribution is closer to 0 than the peak of the N(5,2) distribution. For fifth place the bimodal distribution in Figure 54 fits since it is more accurate than the N(8,2) distribution. This leaves the N(8,2) distribution for last place.

Distribution	RMSE4D	Mean	Median	Optimal rank
Exponential	1.15 (1)	0.88 (1)	0.72 (1)	1
N(2,2)	2.55 (2)	2.10 (2)	1.96 (3)	2
Heavy Tailed $N(0,2) + N(3,5)$	4.30 (3)	2.67 (3)	1.59 (2)	3-4
N(5,2)	5.48 (4)	5.14 (5)	5.31 (5)	3-4
Bimodal $N(1.5,1) + N(8.5,1)$	6.22 (5)	5.04 (4)	4.93 (4)	5
N(8,2)	8.20 (6)	7.93 (6)	7.82 (6)	6

Table 8: Values and relative ranks of different aggregation methods for distributions of error metrics.

From Table 8 it is clear that the ranking based on the RMSE4D values closely resemble the optimal ranking for the considered distributions. It is shown from these results that heavy tails are punished more by the RMSE4D than by the mean or median which is expected. This can be seen when comparing the methods for the *Heavy Tailed* and *Bimodal* as the accumulate error is much higher for RMSE4D

than that of the mean or median. The median method is the least consistent with what is considered the optimal ranking for this scenario and is thus rejected for use when comparing distributions. The mean does generate rankings aligned with the optimal ranking however it fails to favor consistency since it favors the *Bimodal* distribution over the N(5,2) distributions. This indicates that the mean is not suitable to use when aggregating distributions of error metrics as low variance of errors should be encouraged. A benefit of the mean and median however is that the scale of the metric remains the same while the RMSE4D can blow up when large errors are squared. Despite this, the comparison of the three aggregation methods shows that the RMSE4D metric is the superior choice for ranking distributions of error metrics as it favors consistency of error metrics which the mean and the median do not.

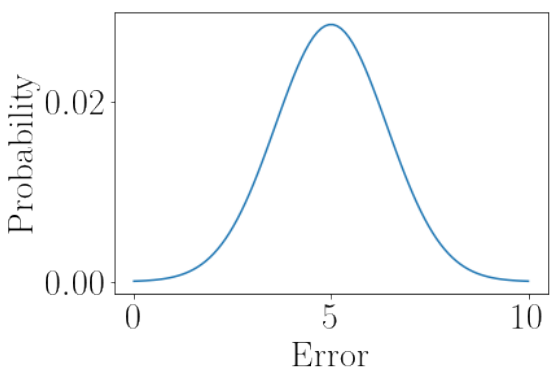

Figure 42: Gaussian

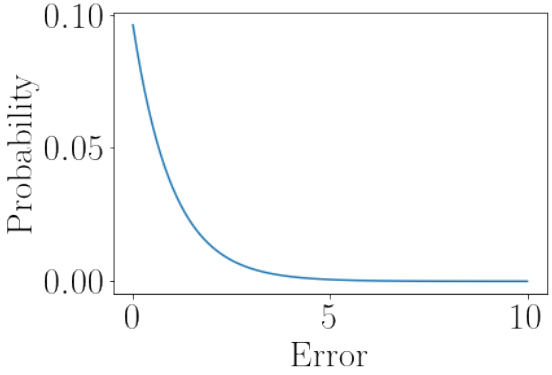

Figure 43: Exponential

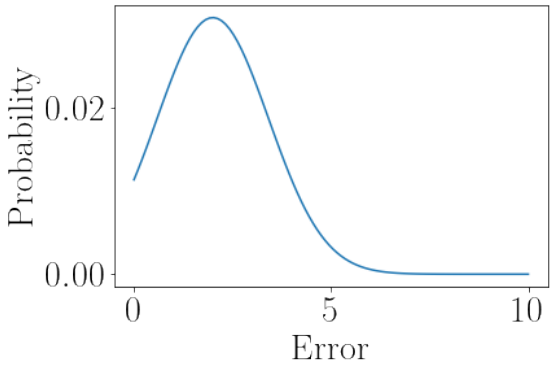

Figure 44: Normal N(2,2)

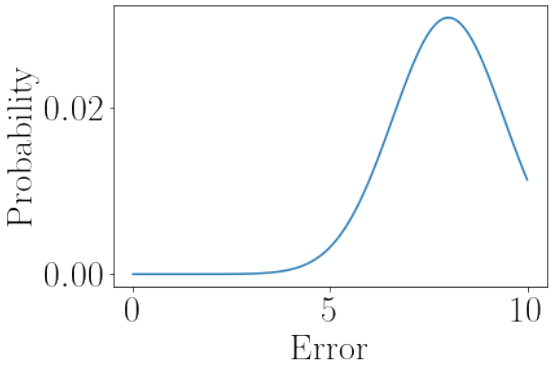

Figure 45: Normal N(8,2)

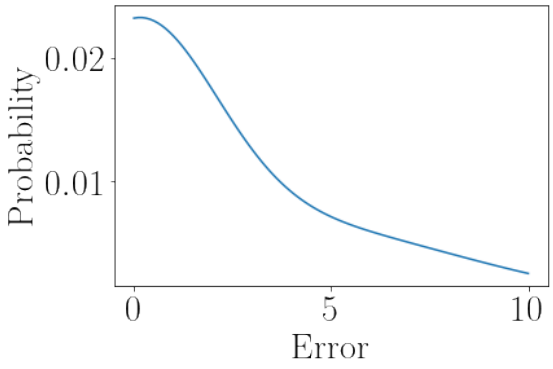

Figure 46: Heavy Tailed

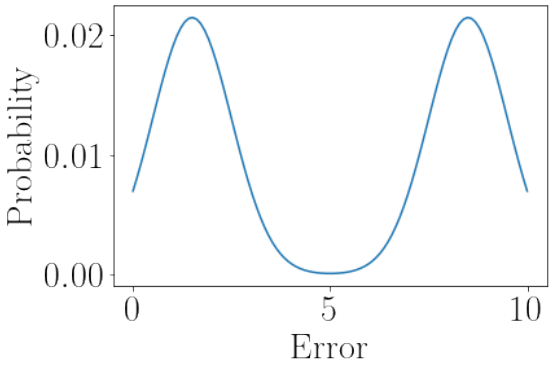

Figure 47: Bimodal

Figure 48: Six possible error metric distributions.

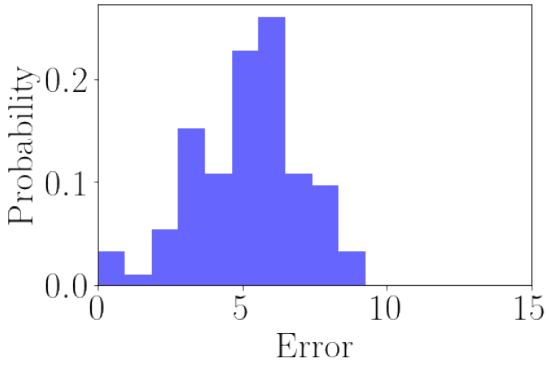

Figure 49: Normal N(5,2)

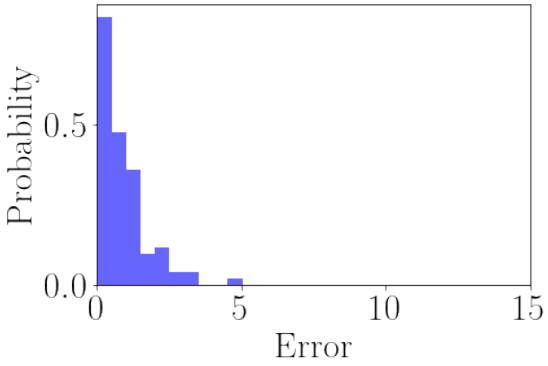

Figure 50: Exponential

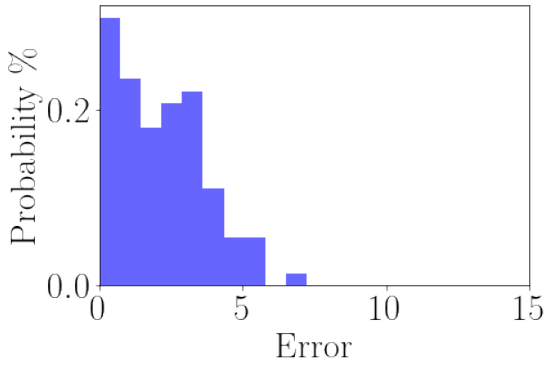

Figure 51: Normal N(2,2)

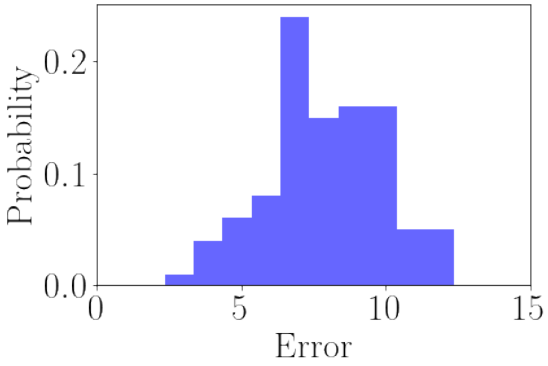

Figure 52: Normal N(8,2)

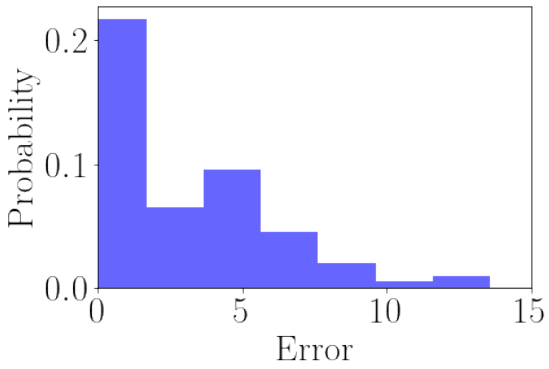

Figure 53: Heavy Tailed

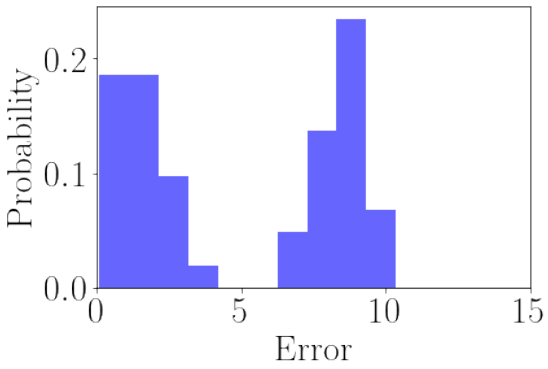

Figure 54: Bimodal

Figure 55: Six histograms from 100 samples of possible error metric distributions.

5 Crayon

In this section Crayon, a Python toolkit for benchmarking ML forecasting models, is introduced. Crayon contains a set of tools for making fair, accurate and reproducible comparisons between forecasting models. Crayon is open sourced and available on GitHub under the MIT license [58].

Crayon is based on the benchmarking architecture presented in chapter 4. Technically reproducible results are made through the use of three core technologies; Runtool configuration files for defining reproducible training jobs, Docker images containing the algorithm to benchmark and the public datasets available through Gluon-TS. In order to make benchmarks statistically reproducible, error metrics generated from benchmarks can be automatically compared through the Kolmogorov-Smirnov 2 sample hypothesis test. Crayon is error metric agnostic, meaning that any error metrics required for a certain domain can be used, thus enabling accurate comparisons between models. Additionally, the RMSE4D aggregation method presented in Chapter 4 is used to rank distributions of error metrics.

Since it is common that machine learning workloads are executed in the cloud, as well as locally, both options are supported by Crayon. Since the runtool only supports executing training jobs in AWS SageMaker a custom backend is implemented for local execution.

5.1 Architecture

This section introduces the overall architecture of Crayon and important modules and concepts are briefly explained. Crayon has three basic buliding blocks, *Algorithm objects Dataset objects* and *Runtool configuration files* (configs). Algorithm objects and Dataset objects is an abstraction simplifying the writing of runtool config file for users and are used internally to generate one or more runtool configuration files. These generated configuration files have a single purpose, to enable technically reproducible benchmarks. In Figure 56 an overview of the five most important modules of Crayon is shown; Benchmarking, Verification, Scoring and the Tuning module. Furthermore, the central use of the BRF described in Chapter 4 is shown.

The Benchmarking functionality showed on top in Figure 56 takes the path to

runtool configuration file and uses the algorithm specified within it to generate four new config files, one for each dataset the algorithm should be benchmarked on. 100 backtests are then executed using each config to generate distributions of error metrics. These backtests can then be executed locally in a sequential manner or in parallel on AWS SageMaker. The recorded error distributions are then stored together with the config file to a BRF to enable statistical and technical reproducibility of the benchmark. After the benchmark is finished, the scoring functionality in Crayon is used to score the distribution of error metrics using the RMSE4D of the error distributions. This score is then ranked against the RMSE4D score of any other benchmarks stored in the BRF which used the same error metric.

Verifying benchmark performance leverages the statistical and technical reproducibility of the benchmarks in the BRF. Since the config used for previous benchmarks is stored in the BRF, a third party can repeat a benchmark as long as the docker image used and the BRF is available. After a benchmark is repeated, the verify module in Crayon can be used to assert statistical reproducibility through applying the kolmogorov smirnov 2 sample test on the two benchmarks error distributions. If the test passed, statistical reproducibility is achieved.

Since optimally tuned models are required for fair comparison as per the discussion in Chapter 4, grid search with repeated runs is implemented in Crayon. Grid search utilizes Algorithm and Dataset objects and a grid of hyperparameters to test. Provided these assets, the grid search generates one runtool config for each hyperparameter configuration and backtests it multiple times before aggregating the runs and evaluating the performance. After tuning finishes, the path to the best config is presented to the user.

This section introduced the high level workings of Crayon, the remainder of this chapter dives further into the features of Crayon with practical examples of various methods and tools.

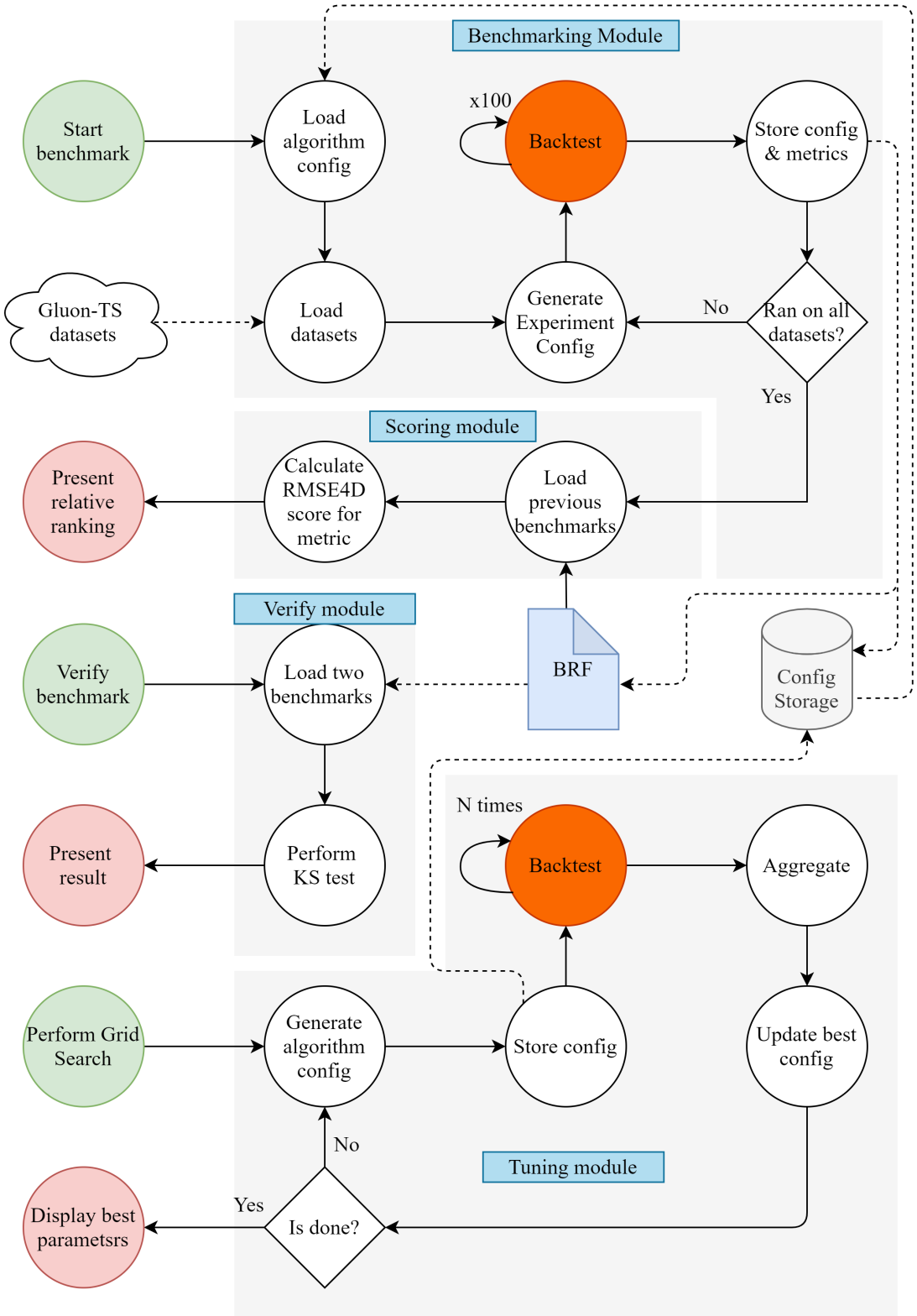

Figure 56: Overview of the functionality in Crayon, start states are marked in green and end states in red, orange states can be executed locally or in the cloud.

5.2 Algorithms

In crayon, the *Algorithms* class contains all the information needed in order to execute an algorithm contained in a Docker image stored either on AWS ECR or on the local machine. *Algorithm* objects are the basic building blocks for performing hyperparameter tuning and for creating runtool configuration files.

An *Algorithm* object contains information about the *image* to run, the *hyperparameters* to use. If the algorithm should be executed on SageMaker, the specific instance it should be run on can also be set using the *instance* parameter. The *Algorithm* class also optionally accepts a dictionary of regexes to the *metrics* parameter. These regexes are used by SageMaker to extract metrics from an algorithms output. Default regexes for Gluon-TS algorithms are available in the *crayon.utils* module. This parameter is not required when executing locally. In Listing 2 an example is shown of how the *Algorithm* class is used.

```
from crayon import Algorithm
Algorithm(
  name="myalgo",
  image="my_image",
  instance="local",
  hyperparameters={...},
  metrics={"abs_error": ... }
)

```

Code Fragment 2: Example of the Algorithm class.

5.3 Datasets

The Crayon *Dataset* class wraps all the data needed to locate a dataset on your local machine or on AWS S3. It only takes two required parameters; the *name* of the dataset & the *path* to the dataset on locally or on AWS S3.

The *path* is a dict which points to the file location(s) of the dataset. If one has a train, test and validation dataset, the items in the *path* will point to these versions of the dataset. Any *meta* information one wish to provide about the dataset in question can be passed through the *meta* parameter. In 3 an example of how to use the Dataset class is presented.

```
from crayon import Dataset
Dataset(
  name="electricity",
  path={
    "train": "file:///datasets/electricity/train/data.json",
    "test": "file:///datasets/electricity/test/data.json",
  },
  meta={...}
)

```

Code Fragment 3: Example of the Dataset class.

5.4 Config Generation

This section discusses the different tools available in Crayon in order to create runtool compatible config files. These config files are a core part of the Crayon functionality and is used in several internal systems. Thus it is important to understand how these are generated and look like. For further information about how the config files function and additional functionality of them, see the section about the runtool (2.9).

Through providing a *Algorithm* object & a *Dataset* object to the *generate_config* function in Crayon, a configuration file compatible with the runtool is generated. The intention of this file is that by providing the config, the image used and the dataset to another person, that person could rerun your experiments with the exact same configuration. Internally Crayon uses the *generate_config* function in order to create tuning and benchmarking training jobs. See 4 for an example.

After executing the code in 4, a config file is stored to the path provided to *generate_config*, in this case the config file is stored to */Users/freccero/Documents/-config.yml*. The config file generated by the code in 4 is displayed in Listing 5.

If one does not have a custom dataset but instead wishes to use a reference dataset for training and evaluating the algorithm on, it is possible to use the datasets in Gluon-TS. The *get_gluonts_datasets* function downloads any number of Gluon-TS datasets onto the local machine and generates a list of Datasets objects which can be passed to the *generate_config* function. Example code for doing so is shown in 6 with the output of the code displayed in 7.

```python
from crayon import Algorithm, Dataset, generate_config

# generate runtool configuration file
config = generate_config(
  Algorithm(
    name="myalgo",
    image="my_image",
    hyperparameters={"epochs": 10},
  ),
  Dataset(
    name="my_ds",
    path={
      "train": "/datasets/electricity/train/data.json",
      "test": "/datasets/electricity/test/data.json",
    },
  ),
  "/Users/freccero/Documents/config.yml",
)
```

Code Fragment 4: Config generation using Crayon

```yaml
my_ds:
  meta: {}
  name: my_ds
  path:
    test: file:///datasets/electricity/test/data.json
    train: file://datasets/electricity/train/data.json
myalgo:
  hyperparameters:
    epochs: 10
  image: my_image
  instance: local
  name: myalgo
```

Code Fragment 5: Generated configurations file.

```python
from crayon import Algorithm, Dataset, generate_config,
    get_gluonts_datasets
import yaml

# generate runtool configuration file
config = generate_config(
  Algorithm(
    name="myalgo",
    image="my_image",
    hyperparameters={"epochs": 10},
  ),
  get_gluonts_datasets(["electricity"]),
  "/Users/freccero/Documents/config.yml",
)

print(yaml.dump(config))

```

Code Fragment 6: Config generation using Crayon with gluonts datasets

5.5 Training

Given a valid runtool config file Crayon uses this file to start training jobs locally or on AWS SageMaker. The provided config file can be either written by hand or generated using Crayon as described in 5.4.

A training job is started by calling the *crayon.run_config* method with the config. The experiment to run is defined using the mathematical operators + and * as described in Section 2.9.

The *crayon.run_config* function returns a *Jobs* object which makes it easy to investigate training results as it stores all metrics reported by the algorithms across all started jobs to a pandas DataFrame.

5.5.1 Local Training

If a config is to be run locally, algorithm defined in the config need to point to an image which is available on the users machine. The image used should be valid for use with AWS SageMaker. Furthermore, any datasets which the algorithm should be trained on should also be available on the local machine. For details of what constitutes a valid SageMaker image, refer to the documentation of SageMaker [56].

In listing 8 an example configuration file for local training is displayed and in 9 code for training the algorithm on the dataset in the config is presented.

By executing the code in Code Fragment 9 Crayon will use the runtool to load

```
electricity:
  meta:
    feat_dynamic_cat: []
    feat_dynamic_real: []
    feat_static_cat:
    - cardinality: '321'
      name: feat_static_cat
    feat_static_real: []
    freq: 1H
    prediction_length: 24
    target: null
  name: electricity
  path:
    test: file:///Users/freccero/.mxnet/gluon-ts/datasets/
    electricity/test/data.json
    train: file:///Users/freccero/.mxnet/gluon-ts/datasets/
    electricity/train/data.json
myalgo:
  hyperparameters:
    epochs: 10
  image: my_image
  instance: local
  metrics: null
  name: myalgo

```

Code Fragment 7: Output when executing Code Fragment 6

```
my_algo:
  image: gluonts:cpu_new
  hyperparameters:
    freq: H
    prediction_length: 24
    forecaster_name: gluonts.model.deepar.DeepAREstimator
  instance: local

my_dataset:
  path:
    train: file:///datasets/electricity/train/data.json
    test: file:///datasets/electricity/test/data.json

```

Code Fragment 8: Config file for running local training jobs

```
from crayon import run_config

jobs = run_config(
  config="config.yml",
  combination="config.my_algo * config.my_dataset",
)

```

Code Fragment 9: Code for running local training jobs using Crayon

the config in 8 and start training jobs locally through the *local_run* method of the runtool. The jobs will be executed sequentially. After the jobs finish, Crayon looks for a file named *agg_metrics.json* or *metrics.json* in the output directory for each job. This file is expected to contain any metrics reported by the algorithm being trained. An example json file is shown in 10 for reference.

5.5.2 Cloud Training

Executing training jobs in SageMaker is done similarly as when starting them locally, but with some alterations to the config file and to the python script.

The config file needs to refer to an image available in ECR of the AWS account that is to be used. Further, the *instance* in the config file must match one of the instance types which SageMaker offers. Any metrics which the algorithm outputs during training such as accuracy metrics or similar must have a corresponding regex under the metrics tag in the config file. This regex is used by SageMaker to parse the algorithm output for metrics. The final change to the config which needs to be done for running on SageMaker is that any datasets needs to be stored in

```
{
  "MSE": 6703174.123610994,
  "abs_error": 37402126.36717987,
  "MASE": 1.62229213532052,
  "MAPE": 0.23736356524438723
}
```

Code Fragment 10: Example of a training job output file such as *metrics.json* or *agg_metrics.json*

an AWS S3 bucket instead of locally.

In addition to the above changes to the config file, three further parameters are required by the *run_config* function in the Python script.

1. *role_arn* - The AWS IAM Role which authorizes Crayon to start training jobs on SageMaker.
2. *bucket* - The AWS S3 bucket where any artifacts of the training job should be stored.
3. *session* - A *boto3.Session* object which Crayon will use to interact with SageMaker.

The IAM Role requires permissions to start training jobs on sagemaker, pull AWS ECR images, and read/write access to the provided S3 bucket. For information about how to create a IAM Role with the proper permissions as well as an S3 bucket please refer to the AWS documentation for these two services [55, 59].

In Listing 11, an example configuration file is displayed which can be used to train on AWS SageMaker and in 12 an example python file is presented which given a config dispatches training jobs to SageMaker using the *run_config* function of Crayon.

5.6 Tuning

In order to use Crayon as a benchmarking tool, it is advised that each algorithm has been suitably tuned to the datasets which it should run on. This is required since comparing a finely tuned model with an undertuned model will result in unfair comparisons being made.

Grid search is a common strategy used when tuning algorithm. When tuning using grid search, one creates for each hyperparameter a list of values which that

hyperparameter can take. Thereafter the algorithm is then run for each combination of these parameters. The hyperparameter combination resulting in the best results is then returned as the optimal hyperparameter combination to use.

Below I present how an algorithm can be tuned using the grid-search functionality in Crayon.

5.6.1 Grid-Search

Grid search is done using the *grid_search* function in the *Crayon.Tuner* module. The *grid_search* function is implemented based on the architecture suggested in 4.4 with some additional features. Since multiple runs of an algorithm generates a small distribution of values, the method used for aggregating these is customizable, per default the mean is taken. Further, since different metrics may optimize towards different values, a custom method for comparing values can be passed, per default lower aggregate errors are better. Each hyperparameter configuration generated by *grid_search* is stored into a runtool configuration file. This simplifes the benchmarking of a tuned algorithm as the generated runtool config file can be used directly by the *crayon.benchmark* method presented in Section 5.7.

An overview of the features of the *grid_search* module is displayed in Figure 57:

Overview of the grid_search function in Crayon

In Code Fragment 13 the signature for the *grid_search* method of Crayon is displayed. Each of the parameters of this method will here be presented and in Code Fragment 14 an example of how to perform grid search locally is presented.

In Code Fragment 13 *changing_hyperparameters* are the hyperparameters which are to be tuned. This parameter takes a dictionary where the keys are the hyperparameters to tune and the value is a list of values which they should take. The *target_metric* parameter selects which metric that the algorithm outputs should be optimized for. Further, the grid search takes a *crayon.Dataset* and a *crayon.Algorithm* (see Sections 5.2 & 5.3). Any hyperparameters defined within the *algorithm* stays the same for each run in the training loop. The *output_dir* determines where the Runtool configuration files created for each hyperparameter combination will be stored.

The *aggregation_function* is used to merge the results if each configuration should be run more than once. Per default this is done is by taking the mean of the recorded values. Through passing a custom function to the *aggregation_function* parameter one can override this behaviour. This function needs to take a list of numbers as a parameters and it needs to return a single value. In Code Fragment 14 the *mean* function is provided from the *statistics* module in the standard library of Python.

The *evaluation_function* is used to determine whether the results of the most recent hyperparameter combination were better than the best seen hyperparameter combination so far. The default behaviour defines that if the new value of the target metric is lower than the best so far, it is better. This makes sense whenever one wishes to minimize the target metric, i.e., an absolute error of 10 is better then an absolute error of 100. However, for other target metrics, different behaviour may be more suitable. Providing a function which takes two parameters, the new value and the old value and compares these to the *evaluation_function* parameter the behaviour is customized. The *run_locally* parameter determines if the jobs should be executed on the local machine or on AWS SageMaker. Any additional parameters passed is forwarded to the *crayon.run_config* function for each training job. Thus, for executing in SageMaker the parameters defined in Section 5.5.2 need to be passed in addition.

In Code Fragment 14 grid search is performed on an image containing gluonts. The algorithm being tuned in Gluon-TS is the DeepAREstimator with the hyperparameters *freq* and *prediction_length* being the same for each training job. The grid search tries to minimizing the average value of the *abs_error* metric over two runs for each hyperparameter combination. The hyperparameters that are tuned are the *epochs* and the *context_length*.

5.7 Benchmarking

One of the key components of Crayon is the benchmarking module. This module compares algorithms by backtesting them on four datasets with different characteristics; *M5*, *M4 Daily*, *Electricity* and *Solar Energy*. These datasets were chosen since they exhibit complimentary characteristics as per the discussion in 4.5.

To benchmark an algorithm via Crayon a config file describing the algorithm needs to be passed. Furthermore, one must pass which target metric should be the target for the benchmark, i.e. absolute error, MASE, MAPE or any other metric reported by the benchmarked algorithm. Crayon then backtests the algorithm on each dataset 100 times in order to build up a distribution of the target metric. The value 100 was chosen as it is in the middle of the boundraries identified in Section 4.6. After the error metric distributions have been generated, RMSE4D described in Section 4.7 is used to compare the errors from the benchmark with previously run benchmarks. A lower value of RMSE4D is better.

As the benchmark is run on multiple datasets, it is important to provide tuned models for each. This is done by appending the dataset name to the algorithm name in the config file so that each algorithm defined in the config file has the structure: $< algorithm\_name > \_ < dataset\_name >$. In Code Fragment 15 an example config file for benchmarking is presented. Note the two hyperparameter

configurations of DeepAR in the config, this indicates that different hyperparameters is used for the electricity dataset.

After the 100 backtests for each dataset has finished, the recorded error distributions are stored to a file on the local machine. This file also contains the *id* of the benchmark, as well as the config used to create the training jobs. Also the *Jobs* object which was generated from the benchmark are stored into this file. The Jobs objects contain the metrics recorded for a certain Job. The file where these values are stored was referred to as the Benchmark Result File (BRF) in Chapter 4. In future versions of Crayon, the BRF could be handled by some centralized storage server instead of locally, however this is out of the scope of the thesis.

5.8 Verifying Benchmark Performance

In 4.3 an architecture for making benchmarks statistically reproducible was presented. In essence, statistical reproducibility of benchmarks can be made through applying a hypothesis test on the error metric distributions generated by the two benchmarks. Since, in Crayon, each benchmark consists of four distributions of error metrics, one for each dataset, four hypothesis tests need to be performed.

The *Crayon.verify* function in Crayon is based on the architecture proposed in 4.3. *Crayon.verify* takes the paths to two BRF files generated by benchmarks. Further, two Benchmark IDs are required in order to identify which benchmarks in the respective files should be verified against each other. The investigation in Section 4.6 concluded that the Kolmogorov Smirnov 2 Sample Test (KS test) is suitable for comparing distributions of error metrics. This test is thus chosen for use in Crayon for verifying benchmarks. Since different metrics are supported by the benchmarking module, the specific metric distribution to verify is required by the *verify* function.

In Code Fragment 17 an example of the *crayon.verify* function is presented. There, two benchmarks from different BRFs are verified against each other using the KS test with respect to the MASE metric. Thus, the KS test is applied to the distributions of the MASE metric for each dataset in turn. If the KS test rejects the two distributions to be samples from the same underlying distribution the algorithms cannot be verified. In Code Fragment 18 example output after executing Code Fragment 17 is presented.

```
my_algo:
  image: 012345678901.dkr.ecr.eu-west-1.amazonaws.com/gluonts
    :2020-11-01
  hyperparameters:
  freq: H
  prediction_length: 24
  forecaster_name: gluonts.model.deepar.DeepAREstimator
  instance: ml.m5.xlarge
  metrics:
  abs_error: 'abs_error\): (\d+\.\d+)'

my_dataset:
  path:
  train: s3://gluonts-run-tool/gluon_ts_datasets/constant/train/
    data.json
  test: s3://gluonts-run-tool/gluon_ts_datasets/constant/test/data
    .json


```

Code Fragment 11: Config file for running training jobs on SageMaker.

```
from crayon import run_config
import boto3

jobs = run_config(
  config="config.yml",
  combination="config.my_algo * config.my_dataset",
  role_arn="arn:aws:iam::012345678901:role/service-role/my_role",
  bucket="my_bucket",
  session=boto3.Session(),
)

```

Code Fragment 12: Code for running training jobs on AWS SageMaker using Crayon

- Performs grid search with both static and changing parameters.
- Each hyperparameter combination can be rerun multiple times to handle non-deterministic algorithms
- Allows passing a custom function for aggregating error metrics from jobs.
- Allows passing of a custom scoring function for evaluating jobs.
- Generates runtool configuration files for each combination of hyperparameters.

Figure 57: Features of the grid search functionality in Crayon.

```
def grid_search(
  changing_hyperparameters: dict,
  target_metric: str,
  dataset: Dataset,
  algorithm: Algorithm,
  output_dir: str = crayon_dir().resolve(),
  aggregation_function: Callable = statistics.mean,
  evaluation_function: Callable = lambda new, old: new < old,
  run_locally: bool = True,
  **kwargs,
) -> GridSearchResults:

```

Code Fragment 13: Parameters of the grid search functionality in Crayon.

```
grid_search(
  algorithm=Algorithm(
    name="deepar",
    image="gluonts_cpu",
    hyperparameters={
      "freq": "D",
      "prediction_length": 7,
      "forecaster_name": "gluonts.model.deepar.DeepAREstimator",
    },
  ),
  dataset=Dataset(
    name="electricity",
    path={
      "train": "file:///electricity/train/data.json",
      "test": "file:///electricity/test/data.json",
    },
  ),
  runs=2,
  target_metric="abs_error",
  aggregation_function = statistics.mean,
  evaluation_function = lambda new, old: new < old,
  changing_hyperparameters={
    "epochs": [1, 5],
    "context_length": [1, 5],
  }
)

```

Code Fragment 14: Grid search running locally.

```
deepar:
  name: deepar
  image: gluonts:cpu_new
  hyperparameters:
    epochs: 1
    freq:
      $eval: $trial.dataset.meta.freq
    prediction_length:
      $eval: 2 * $trial.dataset.meta.prediction_length
    context_length:
      $eval: 2 * $trial.algorithm.hyperparameters.
   prediction_length
    forecaster_name: gluonts.model.deepar.DeepAREstimator
  instance: local

deepar_electricity:
  $from: deepar
  hyperparameters:
    epochs: 2

```

Code Fragment 15: Config file for benchmarking with Crayon. Note that *deepar_electricity* has a different hyperparameter configuration thus these hyperparameters are used when benchmarking the algorithm on the electricity dataset. For all other datasets, the default algorithm definition *deepar* is used.

```
from crayon import benchmark

benchmark(
  algorithm_config="config.yml",
  algorithm_name="deepar",
  benchmark_id="my_benchmark",
  target_metric="MASE",
)

```

Code Fragment 16: Python script for starting a benchmarking run using Crayon.

```python
from crayon import verify

verify(
    id_1="my_benchmark",
    id_1_results_file="BRF_1.yml",
    id_2="someone_elses_benchmark",
    id_2_results_file="BRF_2.yml",
    target_metric="MASE",
)
```

Code Fragment 17: Verifying whether two benchmarks are performed by the same algorithm.

```
    Algorithm verified on dataset electricity
    Algorithm verified on dataset m4_daily
    Algorithm verified on dataset m5
    Algorithm verified on dataset solar_energy
    Passed 4/4 verifications.
```

Code Fragment 18: Output from Code Fragment 17

6 Case Study: Preventing Accuracy Regressions in Forecasting Frameworks

This Chapter investigates how to use the statistical reproducibility of Crayon benchmarks to protect forcasting libraries against accuracy regressions. Specifically, how the *benchmarking* and *verify* functions in Crayon can be used to detect accuracy regression through automated tests. The DeepAREstimator of Gluon-TS will be the focal point of this investigation since it suffered a accuracy regression, which has now been resolved. Thus, distributions can be collected both before, during and after a regression to showcase how these regressions can be detected and how it can be detected when they are resolved.

The DeepAREstimator in Gluon-TS suffered a accuracy regression, between the 7th of May 2020 until 9th of July 2020 for certain values of the *distr_output* hyperparameter was used. The issue arised due to a change in the way that the *gluonts.distribution.NegativeBinomialOutput* was calculated. The issue was not detected until the 15 June 2020, However, the root cause of the issue was not discovered until the 2nd of July after which the bug was patched within a week on the 9th of July [20, 21, 22]. In this case, the time required to detect and resolve the issue was close to two months.

To detect accuracy regression with Crayon, a reference benchmark is required to serve as a ground-truth of the expected performance for an algorithm. When new changes to algorithms or their dependencies happen, a new benchmark should be executed with these changes. These two benchmarks can then be verified using the *crayon.verify* function. If they pass the checks, no accuracy regression has occurred and if the checks failed it indicates that an accuracy regression has occured. When the accuracy regression is fixed a new benchmark can be run as a post-fix benchmark. This post-fix benchmark can be verified towards the ground-truth benchmark to verify if the accuracy is back at the original levels. If a change in accuracy is expected the post-fix benchmark can be used as the new ground-truth for future tests. An overview of this method is shown in Figure 58. This method is designed to easily be integrated into existing testing workflows.

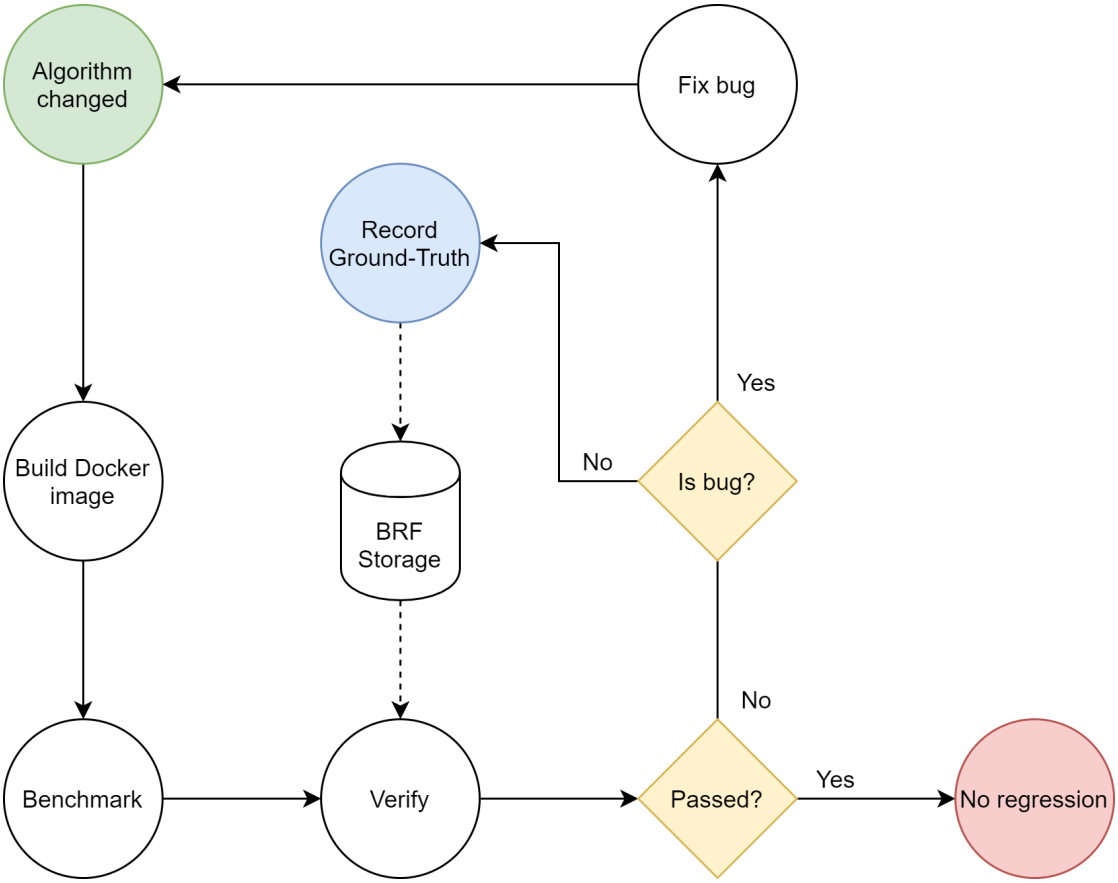

Figure 58: Using Crayon benchmark and verify to protect against accuracy regressions. Initially a ground truth benchmark (in blue) need to be recorded. Therafter, whenever a code change occurs (in green), the algorithm is tested with Crayon through the *crayon.benchmark* & *crayon.verify* functions.

6.0.1 Scope and Limitations

Since the main goal of this thesis is to perform a emprical comparison of forecasting methods, the study made in this section is deliberately small scaled. It serves as a proof of concept of how a benchmarking tool such as Crayon can be applied to other domains such as automated testing during development of time series forecasing algorithms. Furthermore, since executing all of the 400 training jobs required for a complete benchmark removes valuable time from the empirical comparison, the datasets used were the *electricity* and the *m4_daily*.

6.0.2 Methodology

To evaluate Crayon for catching accuracy regressions, three Docker images of Gluon-TS is required. One image containing the code before the regression occured, the second image contains the version of Gluon-TS where the DeepAREstimator suffered an accuracy regression. The third image contains the version of Gluon-TS where the issue was fixed. These three images are hereby referred to as *ground-truth image*, *bugged image* and *post-fix image*. Benchmarks using these three images are then performed and the generated BRFs are stored. Then each benchmark is verified against all other benchmarks with the results summarized into a table. All Docker images are then pushed to the public Dockerhub repository *arangatang/masterthesis* [17].

6.0.3 Result

In Table 9 the results from the study are presented. The results show that the accuracy regression is successfully detected by the KS test for each metric since the verification passes 0 out of 3 times when comparing the *pre-bug* image with the *bugged* image across all metrics. Figure 59 shows violin plots of the RMSE distributions for the three images on the *Solar Energy* dataset. This plot shows that the change in distribution for the RMSE distribution between the bugged and the other images is substantial. Thus it makes sense that the KS test detected this, however, in this scenario a simpler test such as the naive approach investigated in Section 4.6 should suffice.

The data in Table 9 also shows that the *bugged* image fails verification with the *post-fix* image which is expected as the bug should be resolved. However, comparing the *post-fix* image with the *pre-bug* image fails on the *M4 Daily* and the *Solar Energy* datasets. This means that either the performance of DeepAR changed over the course of the 2 months between the patch or that the samples used were too few so that the KS test was unstable.

To investigate this, violin plots of the *pre-bug* and the *post-fix* error distributions were generated. In Figure 60 two of these are presented, one for the MSIS error metric and one for the RMSE error metrics. From these, it seems that the MSIS distribution changed during the two months that the bug was present. Furthermore, it makes sense that the KS metric classified the *pre-bug* metric and the *post-fix* distributions as being sampled from different populations since the mean and variance of the samples differ substantially. However for the RMSE violin plots the difference is not as clear cut.

In total there were 43 commits added to Gluon-TS in the two months when the accuracy regression happened [3]. Three of them are possible causes of the detected change in accuracy between the *pre-bug* and *post-fix* versions. The first

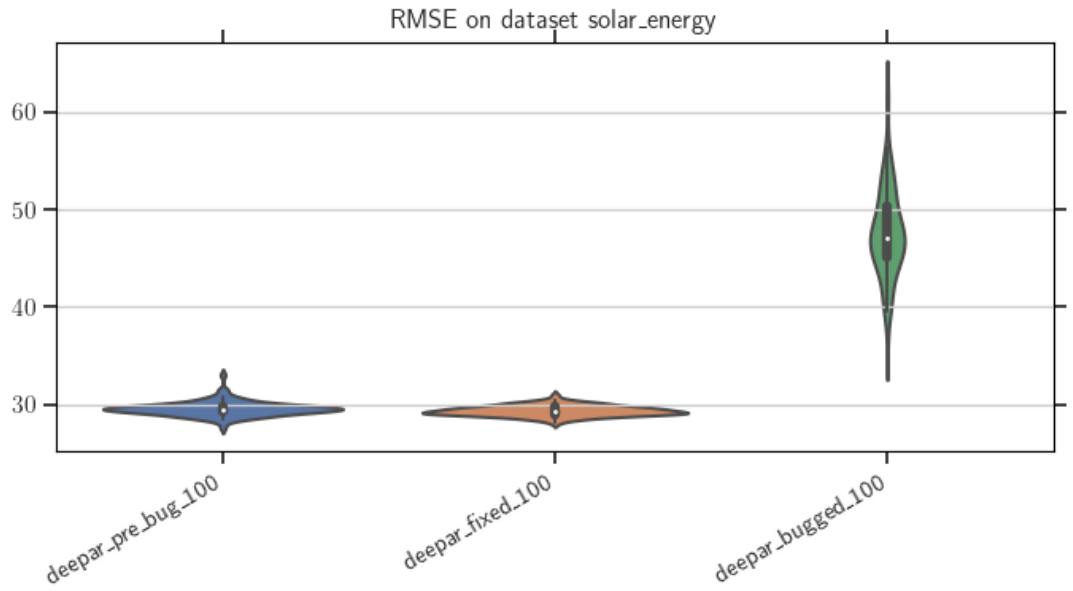

Figure 59: Violin plots of the RMSE error distributions for the three versions of DeepAR on the solar dataset.

of these enabled multiprocessing for backtesting, as discussed in Section 2.5.1 multiprocessing can introduce additional data shuffling which can affect performance. The second change was to how the Negative Binomial was calculated and the third was a change to how MSIS error metrics was calculated. All three of these changes has immediate effect on the error distributions being tested, thus, this indicates that a second accuracy change occurred and that the KS test did not become unstable. Further investigation is needed to verify this further, however due to the time consuming process of benchmarking deepar multiple times more, this is left for future work.

To summarize, the results show that the KS test successfully identified the accuracy regression introduced on the 7th of May. Furthermore, it may have identified a second accuracy change introduced during the time the investigated bug was unnoticed. This result indicates that the benchmark and verification systems in Crayon are suitable for detecting accuracy regressions during development of time series forecasting models in time series forecasting frameworks such as Gluon-TS.

DeepAR version	Pre-Bug	Bugged	Post-Fix
MAPE			
Pre-Bug	3/3	-	-
Bugged	0/3	3/3	-
Post-Fix	3/3	0/3	3/3
MASE			
Pre-Bug	3/3	-	-
Bugged	0/3	3/3	-
Post-Fix	1/3	0/3	3/3
Absolute Error			
Pre-Bug	3/3	-	-
Bugged	0/3	3/3	-
Post-Fix	3/3	0/3	3/3
RMSE			
Pre-Bug	3/3	-	-
Bugged	0/3	3/3	-
Post-Fix	1/3	0/3	3/3
MSIS			
Pre-Bug	3/3	-	-
Bugged	0/3	3/3	-
Post-Fix	1/3	0/3	3/3

Table 9: Number of successfull benchmark verifications for three versions of DeepAR, two without bugs, one with. The benchmarks ran on the *Electricity*, *Solar Energy* and *M4 Daily* datasets and error metric distributions were collected for five different error metrics on each. Optimal results marked in Green, partially optimal results are shown in Orange.

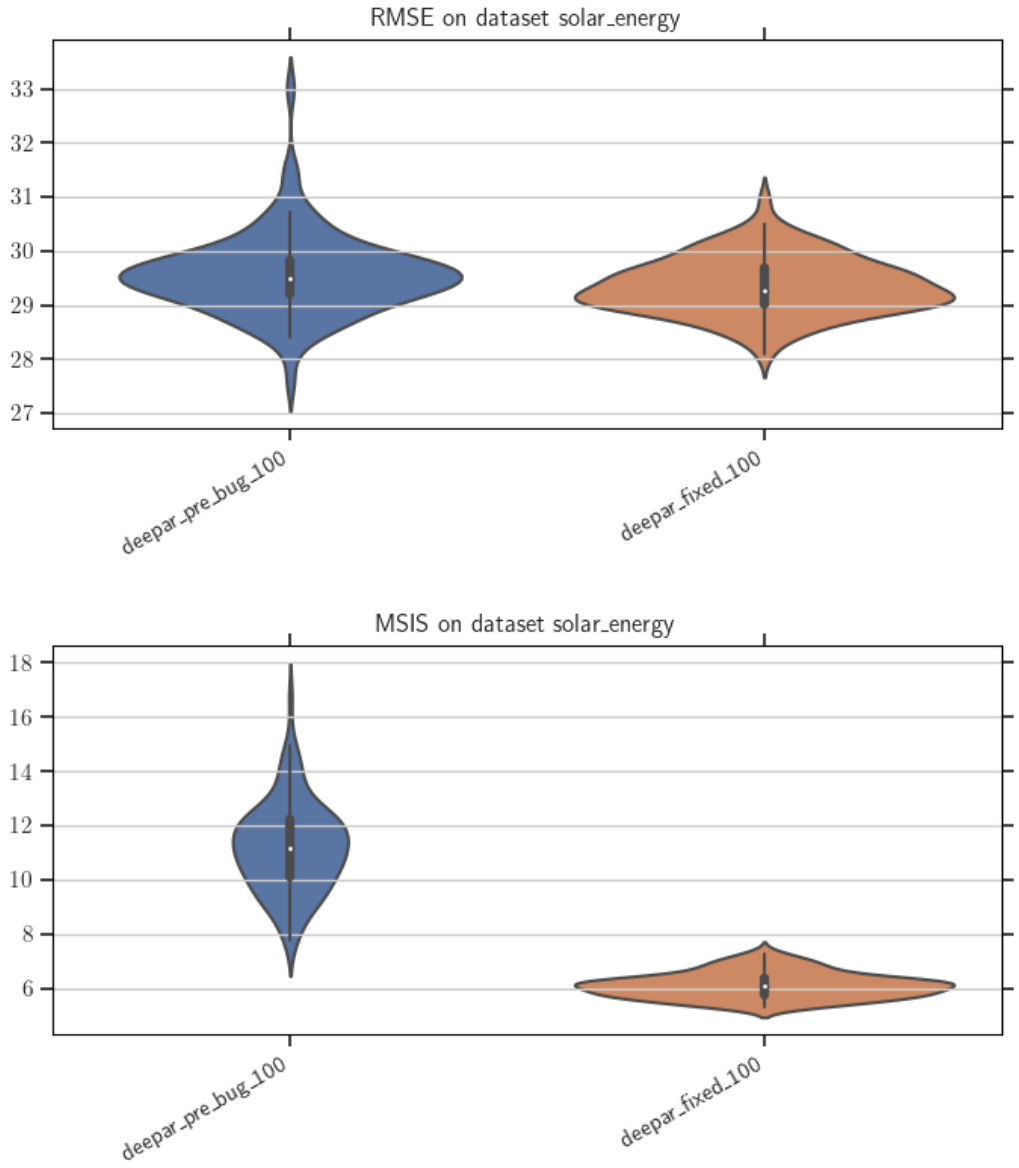

Figure 60: Violin plots of the RMSE and MSIS error distributions on the solar dataset for the *fixed* and *pre-bug* versions of DeepAR.

7 Empirical Comparison of Forecasting Methods

This chapter makes use of the tools and concepts investigated in Chapters 3, 4, 5 to perform an empirical comparison of forecasting methods. Specifically, the tuning and benchmarking capabilities of Crayon are used to enable fair, accurate and reproducible comparisons. The forecasting models compared in this Section are those available in Gluon-TS as of March 18th 2021. In Section 7.1 the methodology employed by this comparison is presented and in Section 7.2 the scope and limitations of the comparison is discussed. The results of the comparison are shown in Section 7.3 and important observations, identifed issues and are thereafter discussed in Section 7.4.

7.1 Methodology

In order to benchmark the forecasting models in Gluon-TS, Docker images containing Gluon-TS have to be built. For this the official dockerfiles available in Gluon-TS are used. After the images are built, methods of speeding up backtesting are investigated. This is done since several thousands of training jobs are required to run, thus, minimizing the time to backtest saves days of training time. Three different methods of speedup are investigated; GPU acceleration, multiprocessing on CPU and using the ultraJSON (ujson) package for serializing and deserializing JSON data. CPU and GPU acceleration is considered as these are the two most most common methods for parallelizing ML workloads and the ujson package is investigated as it is recommended for use by Gluon-TS [3]. If usjon is not installed, the JSON package from the Python standard library is used by instead [3].

After identifying a fast performing configuration the algorithms to benchmark are identified. Due to the size of the comparison, and since the time required to tune and benchmark models differ, a priority list is followed. As the focus of the comparison is on DL based forecasting, those methods utilizing DNNs, CNNs or RNNs have the highest priority. Second highest priority are for recent forecasting models not based on DL. Thereafter, naive methods are chosen as these are quick to evaluate and serve as a good baseline to compare to. After the naive models,

classical models such as arima are chosen since these only serve as a source of comparison but are not the focal point of this thesis.

When the models to benchmark are identified, these are tuned to the four datasets chosen in Chapter 4 using the grid-search functionality in Crayon. The hyperparameter search space is limited for this comparison as several models are to be tuned within a few months on a single machine. Thus in order to ensure that the algorithms are optimally tuned, no limit will be put on the amount of time taken for tuning the algorithms. However, the maximum number of configurations in the grid is set to 300 and the maximum number of different values tested for a single hyperparameter was set to maximum 3. These two limits are imposed as they introduce limits for both models with few hyperparameters and models with multiple hyperparameters as discussed in Section [17]. Furthermore, hyperparameters used for all forecasting models in Gluon-TS share were tuned in the same way for all models for consistency.

Certain information such as the cardinality of a dataset, the prediction length and the frequency of the data are provided as metadata by the datasets in Gluon-TS. These values are used when performing the tuning and benchmarking for this comparison.

When using the Grid Search functionality in Crayon, runtool configurations for each tuning job is stored. When the grid-search has finished the configs of the best hyperparameter configurations are then combined into a single Runtool config file. This file is then passed to the Crayon benchmarking system so that the algorithms can be benchmarked using the optimal hyperparameters for each dataset. The BRF generated by the benchmarks are then uploaded to the public GitHub repository of the masterthesis [18].

The MASE, MAPE, RMSE, Absolute Error and the MSIS error metric are then extracted from the benchmarks in order to compare the forecasting models against eachother. These metrics were chosen in order to showcase how the Crayon Benchmark and the RMSE4D aggregation method works for both scaled and non-scaled metrics. Furthermore, these metrics are some of the most common metrics used in the literature. And when other metrics are used, they are often altered versions of these such as sMAPE or sRMSE. The MSIS error metric is used as it is the only metric which summarises how often a forecasts are within the confidence intervals of the forecasts.

The specification of the computer used to backtest the different models and relevant software used is presented in Table 10.

Component Name	Component Used
CPU	I7-9700K up to 4.9 GHz
GPU	Nvidia Titan RTX
CUDA Version	11.2
GPU Driver Version	460.73.01
RAM	32 GB DDR4, 3600 MHz
Operating System	Ubuntu 20.04.2
Docker Version	20.10.6
Gluon-TS commit	4d1a9a09048b1e0db93de885e9636e4add8f91c6

Table 10: Specification of hardware and software used to perform the benchmarks.

7.2 Scope and Limitations

To limit the scope of this investigation, only algorithms available in Gluon-TS were used. Further, only a subset of these could be used due to the time limit of the thesis. As discussed in Section 3 fair comparisons of advanced algorithms can only be done if the implementations being evaluated are well implemented. Since the algorithms implemented in Gluon-TS are implemented and used by experienced practitioners in the field of forecasting these algorithms are considered to fulfill this criteria. Only algorithms available in Gluon-TS up until commit: *4d1a9a09048b1e0db93de885e9636e4add8f91c6* are used for the comparison, any algorithms added at a later time are not considered [3].

Due to the limited time of the thesis in comparison to the number of algorithms which should be compared, not all algorithms in Gluon-TS can be tuned and benchmarked in time. Further, multivariate algorithms are not compared since Gluon-TS does not offer any multivariate datasets. Further only three multivariate models are available in Gluon-TS, GPVar, GPVar and LSTNet. Any comparisons made with these when executed on the datasets identified in 4.5 would be unfair since only univariate data would be used to train them.

Furthermore, since K-fold cross validation primarily is a method for increasing available test data for small datasets it was not applied to the M4 Daily and M5 datasets since they were the the largest and third largest datasets available in Gluon-TS when looking at available train data. This was primarily an optimization tradeoff since even performing a 2-fold cross validation on the M5 dataset would increase the test time by nearly 50%.

7.3 Result

The initial task of optimizing the Docker images such that they perform well took considerably longer than anticipated. Particularly the use of GPU acceleration was problematic since certain forecasting models such as the *MQCNNEstimator* failed to converge when training. Advice posted on the Gluon-TS Github page suggested that this issue could be resolved by lowering the learning rate, despite this the issue remained [3]. Due to the increased instability, GPU acceleration was deemed unsuitable for the comparison.

The use of the Python package *ujson* showed significant speedup and no negative effects were noticed. Thus, *ujson* is added to the Docker images used for benchmarking and tuning.

Initially, the multiprocessing support when using Gluon-TS within a Docker image was limited. This caused backtesting to fail sporadically. The underlying cause for this was investigated and a pull request solving this was opened [16]. While this made multiprocessing stable, the number of threads used could only be controlled up to 4, if more threads vere requested than four it had no effect as the CPU utilization remained the same. However, executing the models on four cores instead of one did improve throughput significantly and was thus utilized for the benchmarks.

Most of the algorithms in Gluon-TS can be executed through the same Docker image. However, the naive forecasting algorithms, the Prophet forecaster, and the classical models had additional dependencies. Thus in total four Docker images were built. These images are available on the *arangatang/masterthesis* repository on Dockerhub [17].

Of the 20 algorithms presented in Section 2.8.1 four does not require tuning; Prophet, Naive Seasonal, Naive 2 and the Theta from the RForecastPredictor. Since multivariate algorithms are excluded from the study, the GPVAREstimator, DeepVAREstimator and the LSTNET estimators are not tuned. In order to tune the remaining 13 models, 920+ configurations were executed on 4 datasets with 3 repetitions per hyperparameter configuration. Thus, more than 11000 training jobs were executed over the course of four months. An overview of the results from the tuning is presented in Table 11.

Of the 13 models which required tuning, 7 were tuned without major issues, however for five of the algorithms; Gaussian Process (GP), DeepState, WaveNet, NPTS and Seq2Seq harder issues arose when executing on CPU. The issue with the GP Estimator stemmed from how the *cardinality* hyperparameter was passed. For all other algorithms in Gluon-TS the cardinality is passed as a list of integers but the GP Estimator requires a single integer to be passed. This is easily fixed for three out of the four datasets as the cardinality stored in the datasets metadata only contained one value. However, the metadata of the M5 dataset contained

Algorithm	Electricity	Solar Energy	M4 Daily	M5	Electricity (GPU)
DeepAR	X	X	X	X	X
DeepFactor	X	X	X	X	X
CanonicalRNN	X	X	X	X	X
SimpleFeedForward	X	X	X	X	X
N-BEATS Ensamble	X	X	X	X	X
Gaussian Process	X	X	X	F	X
MQCNN	X	X	X	X	F
MQRNN	X	X	X	X	X
Transformer	X	X	X	X	X
DeepState	M	-	-	-	M
WaveNet	F	-	-	-	F
NPTS	F	-	-	-	F
Seq2Seq	-	-	-	F	-
Prophet	U	U	U	U	U
Naive Seasonal	U	U	U	U	U
Naive2	U	U	U	U	U
Theta	U	U	U	U	U

Table 11: Tuning results of 16 forecasting algorithms in Gluon-TS. Legend: X (green) finished successfully, M (orange) memory leak, F (red) failed, U (blue) tuning not required.

multiple values in the cardinality. Thus, the GP estimator failed to execute for the M5 dataset. WaveNet failed due to multiprocessing issues, since considerable time had been spent in fixing the multiprocessing system for Gluon-TS when Dockerised, debugging the WaveNet algorithm and tuning it was considered to take too much time, investigating it further was thus put on hold. The NPTS algorithm failed due to compatibility issues with the Gluon-TS shell, fixing this was considered outside the scope of the thesis. The Seq2Seq estimator was one of the last estimators to be tuned, when it failed not much time was left to tune the models thus, the Seq2Seq estimator was excluded from the comparison. The tuning of DeepState was cancelled when all of the available 32 GB of RAM and 32 GB of swap had been utilized. This happened before the first training epoch was finished. Thus it is unclear whether the DeepState estimator suffered a memory leak or if it had unusually high memory requirements compared to the other algorithms in Gluon-

TS. An overview of the tuning results is presented in Table 11.

After the tuning of the algorithms finished, the 13 remaining algorithms were benchmarked through crayon. This resulted in 5600 training jobs being executed over the course of two months. Tables; 14, 15 and 13 present the results of these for the MASE, MAPE, RMSE, MSIS and Absolute error metrics. These error metrics are aggregated using the RMSE4D methods as described in Section 4.7. In Table 12 the average rank of each benchmark is presented to summarize Tables 14, 15 and 13.

Algorithm	MASE	MAPE	RMSE	Absolute Error	MSIS	Mean Rank
DeepAR	3.25	6.75	3.00	4.00	2.25	3.80
N-BEATS	2.50	5.50	3.75	2.25	6.00	3.90
Transformer	3.50	6.75	4.25	4.50	2.75	4.25
SimpleFF	4.75	5.75	4.50	5.50	2.25	4.45
MQCNN	6.00	5.50	4.50	4.75	11.00	6.15
C-RNN	6.75	8.00	6.50	6.50	4.25	6.30
S-Naive	6.75	4.75	7.75	6.25	7.75	6.30
Thetaf	6.50	2.75	5.75	6.50	7.50	6.90
Naive2	8.50	6.50	10.25	8.50	8.50	8.10
DeepFactor	10.00	8.75	9.75	10.00	7.00	8.60
MQRNN	9.25	8.00	9.25	9.00	11.00	8.95
GP	10.75	8.75	10.25	10.75	9.00	9.55
Prophet	12.00	12.00	11.00	12.00	8.25	11.05

Table 12: Mean ranks of the benchmarks for the MASE, MAPE, RMSE, Absolute Error and MSIS error metrics.

As presented in Table 12 DeepAR is the algorithm with the best average ranking across all datasets and error metrics. Since DeepAR automatically applies categorical features to the dataset it may have gained an upper hand in this comparison since this data is not part of the datasets used. The source paper of DeepAR evaluates DeepAR on the Electricity dataset, however, the metric used is a modified version of RMSE known as nRMSE which makes it impossible to compare the results as the metrics have different scales. However in the M5 competition, a modified version of DeepAR was the third best performing forecasting algorithm. This reinforces the findings from this research since DeepAR was the second best performer for the M5 dataset on three out of the five error metrics, Tables 14 & 15 and the fourth best on the MSIS metric, see Table 13.

The N-BEATS Ensemble Estimator is the second best performer with an average rank between 2 - 6 across all tested metrics according to Table 12. While it generally performed well, its performance on the M4 Daily dataset was average

to poor (between 4-10). This implies that N-BEATS may be unsuited for heavily trended data and may thus benefit from detrending the data beforehand. In the original paper of N-BEATS it is evaluated on the M4 dataset, however not only on the *M4 daily* data as has been done here but on the complete M4 dataset. This in combination with them utilizing a modified MAPE metric (sMAPE) for ranking their results makes comparing results hard. They do however show that N-BEATS perform better than Theta on the M3 dataset. This can be seen in the results from this benchmark as well for the Electricity and Solar Energy datasets. For the M4 Daily dataset, this is however not the case as Theta performs better than N-BEATS for all metrics. Not only is theta better than N-BEATS on the M4 Daily, both of the Naive approaches also outperform N-BEATS and all other DL models for the Absolute Error, MASE and MAPE metrics on the M4 Daily dataset, see Tables 14 & 15.

The Transformer forecaster is the third best ranked algorithm and has the 2nd best MSIS score (see Table 13) of all tested algorithms after DeepAR and Simple Feed Forward which is the fourth best performing algorithm. This indicates that these two algorithms sacrifice specificity of forecasts in exchange for better coverage of the probability intervals. I.e. their forecasts are less often spot on but rarely very wrong. It is surprising that SimpleFF ranked so high in this comparison as it outperformed more sophisticated DL approaches like the MQCNN, MQRNN and DeepFactor with its simple neural architecture. Furthermore, the performance of the SimpleFF also outperformed the C-RNN architecture despite C-RNN containing LSTM cells which tends to be better for temporal data such as time series. Possibly, C-RNN overfit to the datasets when tuning as the hyperparameters configuration found when tuning were larger neural nets with more cells for the smaller datasets and smaller nets for the larger datasets. For the datasets where C-RNN performed worst, *M4 Daily & Electricity* 210 cells were used compared to 100 for *M5 & Solar Energy*.

MQCCN was the fifth best performing algorithm, with a mean rank of 6.15. It should be noted that the MSIS error was not reported by Gluon-TS for either MQCNN or MQRNN and that their ranks suffered as a result, see Table 13. This is counterintuitive as both of these produce probabilistic forecasts which means calculating the MSIS should be without issues. Recalculating the mean rank for these two algorithms without MSIS, i.e., using the data in Tables 14 & 15 would cause MQCNN to have a mean rank of 5.19 and MQRNN 8.875 instead of 8.95, thus, their overall ranks would remain the same in this comparison. One possible reason why these two models are not ranked higher is due to the lack of exogenous data when training the model. In the original paper of these algorithms, the authors manually added calendar features such as holidays and which day of the week each datapoint in the datasets were recorded on [70].

The Theta Forecaster was developed to function well for the M3 dataset, thus, it is not surprising that it performs the best of all tested models on the M4 dataset as the M4 dataset is similar in composition to the M3 dataset but larger [64]. The performance of Thetaf was however in the bottom 50% for the other datasets, which indicates that while it performs well for datasets having high trend and low seasonality, such as the M4, it does not generalize well to datasets with other characteristics. Particularly this is seen for highly seasonal datasets such as the Solar Energy dataset and the Electricity datasets where theta performs in the bottom 50%.

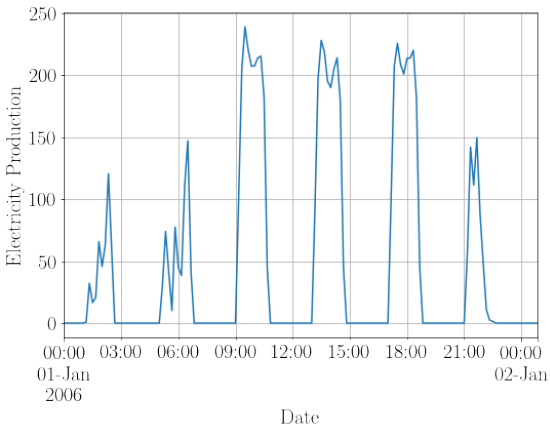

Figure 61: Solar Energy timeseries with frequency 10 min.

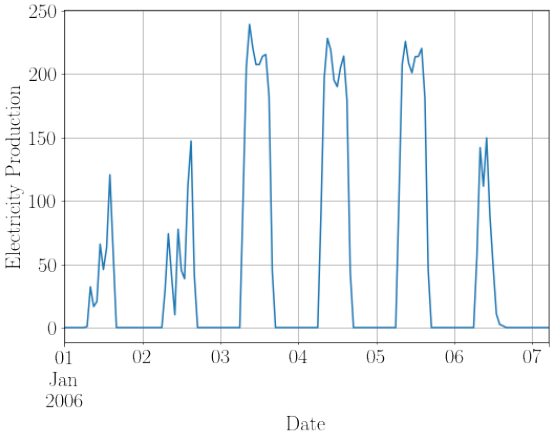

Figure 62: Solar Energy timeseries with frequency 1 hour.

The S-Naive forecaster performed well on the Electricity and the M4 daily dataset with average ranks of 3.4 and 3.6 respectively. However for the Solar Energy dataset the average rank across all metrics was 8.6. This does not make sense as the solar energy dataset has the highest strength of seasonality of all datasets. Thus sampling values from previous seasons as is done in the S-Naive forecaster should bring competitive performance at least on par with the performance of S-Naive on the other datasets. After investigating this further an issue with the Solar Energy dataset in Gluon-TS was identified. The frequency the dataset is sampled with is 10 minutes according to the metadata. However, this could be a mistake and that the data has a frequency of 1H. In Figure 61 a time series from the dataset is shown plotted using a frequency of 10 minutes. If the true frequency of the data would be 10 minutes, then the power production of the solar panels spikes for two hours and then produces zero energy for two hours. Since solar panels produce power when the sun is out during the day and not when it is night, these two hours cycles would mean that the day is two hours long. By setting the frequency to *1H* instead of the original *10min* the timeseries is properly aligned as

in Figure 62. This is a likely cause as to why the Seasonal Naive performed poorly on the Solar Energy dataset. Since the Thetaf forecaster deseasonalizes each time series when it is run, the ranking of this could have been affected as well. The faulty frequency to the Solar Energy dataset also likely caused the automatic feature engineering in DeepAR to add calendar features to the wrong parts of the data, i.e. marking datapoints sampled in April to be sampled in December.

The second naive model, Naive2, crashed on the Electricity dataset due to it utilizing multiplicative decomposition of time series which is not suitable for time series data close to 0. The API for this model in Gluon-TS does not support additive decomposition, thus not much could be done to fix this. The average rank of the Naive2 method was 8.10 which makes it that fourth worst performing method of the ones tested.

The low rank of DeepFactor contradicts the results from its original paper as there, DeepFactor, outperformed DeepAR on average while in Table 12 DeepAR outperforms all other algorithms. This further enforces the benefit of automatically adding exogenous variables such as calendar days and holidays to the data like DeepAR does. The original paper of DeepFactor also compares it against MQRNN and Prophet, their findings do align with the results of this comparison since DeepAR has a better average rank than both. However, Prophet did not have time to finish the benchmark on the *M5* and *Electricity* datasets. Thus, comparing DeepFactor with Prophet only by looking at the mean ranks across all dataset is not fair as the average rank of Prophet is inflated by the last place ranks caused by not being ranked on a dataset. If Prophet would not have been the last algorithm to be benchmarked for this comparison it would likely have produced some error metrics for these datasets. Thus, Prophet is a special case in this comparison and should be compared only on those datasets it finished on since it was not responsible for not concluding the benchmarking on time. Comparing the mean ranks of DeepFactor and Prophet across all metrics for the Solar Energy and the M4 Daily datasets tells a similar story where DeepFactor recieves an average rank of $10 + 11.5 + 10 + 9.5 + 9 = 10$ and Prophet receives the mean rank $12 + 12 + 12 + 10 + 6.5 = 10.5$ see Tables 14, 15 and 13 for the values used in this calculation. Thus, for this comparison, DeepFactor is only marginally better than Prophet.

The Gaussian Process (GP) forecasting method was ranked second last, with Prophet in the last place. The GP method was likely suboptimally tuned as the kernel used for creating the Gaussian Processes could not be specified through the Gluon-TS shell when tuning. This may have led to a subpar kernel being used for the benchmark.

Algorithm	Electricity	Solar Energy	M4 Daily	M5	Mean rank
			MSIS		
DeepAR	6.44 (1)	8.55 (2)	43.13 (2)	32.58 (4)	2.25
SimpleFF	10.64 (3)	14.78 (3)	36.33 (1)	26.49 (2)	2.25
Transformer	6.85 (2)	6.41 (1)	49.15 (3)	35.38 (5)	2.75
C-RNN	21.96 (4)	46.74 (6)	105.21 (4)	31.41 (3)	4.25
N-BEATS	33.35 (5)	46.50 (5)	137.24 (8)	57.74 (6)	6.00
DeepFactor	479.13 (9)	53.17 (7)	928.47 (11)	20.78 (1)	7.00
Thetaf	47.02 (7)	97.08 (11)	130.49 (5)	69.13 (7)	7.50
S-Naive	35.25 (6)	97.07 (9)	131.14 (7)	81.28 (9)	7.75
Prophet	nan (10)	20.84 (4)	166.84 (9)	nan (10)	8.25
Naive2	nan (10)	97.07 (10)	131.14 (6)	81.28 (8)	8.50
GP	126.45 (8)	84.13 (8)	185.82 (10)	nan (10)	9.00
MQRNN	nan (10)	nan (12)	nan (12)	nan (10)	11.00
MQCNN	nan (10)	nan (12)	nan (12)	nan (10)	11.00

Table 13: Benchmark results of 100 runs aggregated using RMSE4D with respects to the MSIS error metric. Ranks within a dataset shown in parentheses.

7.4 Discussion

The overall result of this empirical comparison of forecasting methods showed that DeepAR, N-BEATS and the Transformer forecasting methods are the best performing forecasting methods out of the 13 investigated. Furthermore, statistical and naive models were the strongest performer for the *M4 Daily* dataset but underperformed compared to DL based forecasting methods on the *Solar Energy, M5* and *Electricity* datasets. Essentially, the *"No Free Lunch"* theorem is demonstrated by these results as none of the benchmarked algorithms were the best on all datasets for a single metric or on all metrics for a single dataset.

A learning from this result is that exogenous variables should be added to atleast some datasets used by Crayon when benchmarking as this would improve the fairness when comparing the methods. Perhaps multiple versions of each dataset could exists, both with and without preproccessing and exogenous variables. It should be noted that ease of achieving strong performance for a forecasting algorithm also should be prioritized by the creators of the algorithms. This goes along the philosophy of DeepAR and Prophet which both focused on making the algorithms easy to use.

The findings of the M4 competition that DL methods perform worse than simple models are partially disproven. For the M4 Daily dataset their findings are accurate as only one DL method was in the top five best performing algorithms. However, for the other datasets which were not used in the M4 competition, the claim that simple models are superior is disproven as both the naive models and Theta consistently perform worse than the DL models across all metrics which also validates the findings of the M5 competitions. However, it would be beneficial to also benchmark models from other frameworks than Gluon-TS, specifically Gradient Boosting Tree based models as these were overall best performing algorithms in the M5 competition.

Of the benchmarking systems discussed in Section 2.6 only the Libra benchmark was specifically targeted to time series forecasting. However in comparison to Crayon, Libra focused on generating diverse datasets for local models. Thus, the benchmaring capability offered by Crayon complements Libra by instead offering datasets and tooling to reproducibly benchmark global as well as local models. It would be interesting to extend the benchmarks performed in this comparison with benchmarks on the datasets offered by Libra to identify how global models such as DeepAR performs with the disparate timeseries of the Libra datasets.

The high accuracy achieved on average by the N-BEATS model show the benefit of ensambling NN models together. While this does increase the time to train and backtest significantly, this is an acceptable tradeoff for applications where training time does not matter. However, a possible improvement of Crayon would be to take the time taken for backtesting into account when ranking models. A possible improvement of the benchmarking system leveraging this would be to average multiple different RMSE4D ranks together with different weighting applied. For example for some domains the MSIS coverage is more important than i.e. the MASE, and for other domains, the execution time is more important. By associating each metric with a weight, suitable scores could be calculated for different domains. Investigating how such a weighting scheme could be designed would be interesting future work to improve the Crayon benchmark.

The findings of Pritzche in *Benchmarking of Classical and Machine-Learning Algorithms (with special emphasis on Bagging and Boosting Approaches) for Time Series Forecasting* aligns with the findings of this empirical comparison in two ways; the poor performance of the Gaussian Process method and that ensambling is a good alternative to use when time allows. However, the results of this comparison contradicts the poor performance of NN models over classical methods that Pritzche reported. While the C-RNN model which is the most similar model to the NN models used by Pritzche did perform subpar compared to the other benchmarked models this is a very basic NN. When instead comparing state of the art models such as DeepAR it is clear that DL models do not perform worse than

classical and naive models except for on specific datasets.

Using RMSE4D for aggregating distributions of error metrics was successully applied in this comparison in order to make it possible to rank algorithms against each other. Furthermore, it is powerful since scaled metrics retain their capability of being compared across datasets. I.e. RMSE4D of MASE for Electricity can be compared with RMSE4D of MASE for M5. However, a limitation of the RMSE4D is that the source metric cannot be compared. For example, RMSE4D of MASE cannot be compared with a single MASE metric as the RMSE4D MASE has been scaled. This limits the ease of generalizing the results which, for example, the mean does not. Despite this, as shown in Section 4.7 the RMSE4D is superior than the mean for comparing different distributions of error metrics with each other as it is biased towards algorithms having low variance in errors.

If a researcher designed a new algorithm and wished to compare it with the 13 algorithms evaluated in this thesis it could be time and cost prohibitive to execute the 15 000+ backtests required. While the time required could be lessened by using a more sophisticated tuning algorithm than grid search or by using a compute accelerator such as a GPU, it could still be cost restrictive. A method of easing the computational demands of reproducing results with Crayon could be by offering a cloud based service for storing and validating BRFs. By storing the BRF and any reproduction attempts of each benchmark, findings would only need to be reproduced once by a third party. Thereafter, researchers would not need to rerun the benchmarks as the benchmark already is verified. Instead the distributions generated for a benchmark could be utilized in other research papers. Furthermore, this type of collaborative research could help advance the speed of innovation in time series forecasting research as fewer overall benchmarks would be required. Crayon was developed with this in mind, where it could evolve into a cloud based collaborative & competitive benchmarking service for reproducible benchmarking.

Algorithm	Electricity	Solar Energy	M4 Daily	M5	Mean rank
MASE					
N-BEATS	0.83(3)	1.16(2)	3.43(4)	1.44(1)	2.50
DeepAR	0.75(1)	1.17(3)	3.81 (7)	1.48(2)	3.25
Transformer	0.82(2)	1.13(1)	4.26 (8)	1.50(3)	3.50
SimpleFF	0.92 (5)	1.27(4)	3.48 (5)	1.54 (5)	4.75
MQCNN	1.22 (7)	1.50 (5)	3.51 (6)	1.55 (6)	6.00
Thetaf	1.18 (6)	2.43 (11)	3.26(1)	1.73 (8)	6.50
S-Naive	0.88(4)	2.43 (10)	3.28(2)	2.03 (11)	6.75
C-RNN	2.24 (8)	1.60 (6)	4.31 (9)	1.52(4)	6.75
Naive2	nan (12)	2.43 (9)	3.28(3)	2.03 (10)	8.50
MQRNN	11.19 (10)	1.71 (7)	57.94 (13)	1.58 (7)	9.25
DeepFactor	12.02 (11)	1.84 (8)	23.64 (12)	1.85 (9)	10.00
GP	3.18 (9)	2.45 (12)	4.69 (10)	nan (12)	10.75
Prophet	nan (12)	3.26 (13)	11.96 (11)	nan (12)	12.00
MAPE					
Thetaf	0.18 (6)	1.00(1)	0.04(3)	0.65(1)	2.75
S-Naive	0.14(4)	1.00(3)	0.04(1)	0.88 (11)	4.75
N-BEATS	0.13(3)	2.84 (7)	0.05 (5)	0.82 (7)	5.50
MQCNN	0.22 (7)	2.15 (6)	0.05(4)	0.79 (5)	5.50
SimpleFF	0.14 (5)	2.96 (8)	0.05 (6)	0.79(4)	5.75
Naive2	nan (12)	1.00(2)	0.04(2)	0.88 (10)	6.50
DeepAR	0.10(1)	3.29 (10)	0.05 (7)	0.87 (9)	6.75
Transformer	0.10(2)	3.27 (9)	0.06 (8)	0.86 (8)	6.75
C-RNN	0.30 (8)	4.92 (12)	0.06 (9)	0.79(3)	8.00
MQRNN	1.01 (11)	1.03(4)	0.42 (13)	0.82 (6)	8.50
DeepFactor	0.99 (10)	4.09 (11)	0.20 (12)	0.70(2)	8.75
GP	0.34 (9)	1.09 (5)	0.06 (10)	nan (12)	9.00
Prophet	nan (12)	8.22 (13)	0.15 (11)	nan (12)	12.00

Table 14: Benchmark results of 100 runs aggregated using RMSE4D with respect to the MASE and MAPE metrics. Ranks within a dataset shown in parentheses.

Algorithm	Electricity	Solar Energy	M4 Daily	M5	Mean rank
Absolute Error					
N-BEATS	8869733(1)	336976(2)	11201583 (5)	811248(1)	2.25
DeepAR	9327671(4)	339306(3)	12135357 (7)	844784(2)	4.00
Transformer	9263741(3)	328560(1)	13681491 (9)	869196 (5)	4.50
MQCNN	13805770 (7)	432799 (5)	11070810(4)	856588(3)	4.75
SimpleFF	9531230 (5)	369138(4)	11234445 (6)	904818 (7)	5.50
S-Naive	8962995(2)	708874 (10)	10701169(2)	1152816 (11)	6.25
Thetaf	13089580 (6)	708907 (11)	10512352(1)	912691 (8)	6.50
C-RNN	29287248 (8)	465076 (6)	13542748 (8)	861390(4)	6.50
Naive2	nan (12)	708874 (9)	10701169(3)	1152816 (10)	8.50
MQRNN	121965994 (10)	499512 (7)	205836216 (13)	880390 (6)	9.00
DeepFactor	128561908 (11)	531101 (8)	72334786 (12)	1025815 (9)	10.00
GP	38069033 (9)	715938 (12)	15070775 (10)	nan (12)	10.75
Prophet	nan (12)	950621 (13)	40508217 (11)	nan (12)	12.00
RMSE					
DeepAR	1752.89 (5)	30.81(3)	637.84(2)	2.28(2)	3.00
N-BEATS	1177.71(2)	30.53(2)	748.49 (10)	2.21(1)	3.75
Transformer	1399.50(4)	29.55(1)	677.32 (5)	2.53 (7)	4.25
SimpleFF	1229.84(3)	35.58(4)	672.66(3)	2.79 (8)	4.50
MQCNN	1850.22 (6)	39.29 (5)	672.82(4)	2.31(3)	4.50
Thetaf	1936.66 (7)	62.51 (10)	630.09(1)	2.43 (5)	5.75
C-RNN	4629.80 (8)	41.28 (6)	745.09 (8)	2.39(4)	6.50
S-Naive	1139.93(1)	62.52 (13)	705.42 (6)	3.30 (11)	7.75
MQRNN	13405.46 (10)	45.92 (8)	5443.89 (13)	2.53 (6)	9.25
DeepFactor	13505.91 (11)	42.26 (7)	2192.18 (12)	2.97 (9)	9.75
GP	6113.39 (9)	62.52 (11)	747.68 (9)	nan (12)	10.25
Naive2	nan (12)	62.52 (12)	705.42 (7)	3.30 (10)	10.25
Prophet	nan (12)	53.51 (9)	1464.99 (11)	nan (12)	11.00

Table 15: Benchmark results of 100 runs aggregated using RMSE4D with respect to the Absolute Error and RMSE metrics. Ranks within a dataset shown in parentheses.

8 Conclusion and Future Work

This thesis investigated methods for performing fair, accurate and reproducible comparisons of forecasting methods and applied these to compare 13 forecasting methods on four public datasets. In Chapter 3 the terms, fair, accurate and reproducible were defined and criterias for fullfilling them were established. These criterias were used in Chapter 4 to design a system for enabling fair, accurate and reproducible comparisons. Additionally, three studies were performed in Sections 4.5, 2.10 and 4.7 to identify appropriate datasets to use, suitable hypothesis tests for reproducible results and how distributions of error metrics could be accurately compared. The datset analysis identified the M4 Daily, Electricity, Solar Energy and M5 datasets as suitable datasets as they have diverse characteristics in terms of trend and seasonality while the Kolmogorov-Smirnov test was identified as the most suitable statistical test to use for statistical reproducibility. In Section 4.7 the RMSE4D metric was defined and evaluated against other aggregation methods on simulated data to determine how distributions of error metrics could be accurately compared. Chapter 5 introduced Crayon, an open source benchmark for fair, accurate and reproducible comparisons of time series forecasting methods. In Section **??** the benchmarking and verification methods in Crayon was presented showing how the technical and statistical reproduciblity of Crayon benchmarks could be used to protect forecasting methods and frameworks from accuracy regressions. Not only was the intended accuracy regression successfully identified, another possible accuracy regression was found to have been introduced during the time of the first accuraccy regression. In Chapter 7 an empirical comparison of forecasting methods took place where 13 state of the art forecasting methods were tuned and benchmarked using Crayon on the four datasets identified in 4.5. The findings of this Comparison showed that DeepAR, N-BEATS and the Transformer estimators were the best performing models when ranking them based on the MASE, MAPE, MSIS, RMSE and Absolute Error metrics. However, for the M4 Daily dataset the Naive and Statistical models such as Theta outperformed all NN and hybrid approaches. This demonstrated that the "No Free Lunch" theorem holds as no single method was superior at all times.

Future improvements of the Crayon benchmark would be to add additional datasets such as those offered by Libra to enable fair comparisons to local models. Furthermore, exogenous variables such as day of the week or public holidays should be added to the datasets used when benchmarking. Further, while the RMSE4D

metric introduced in this thesis is suitable for comparing distributions of error metrics it would be interesting to extend it such that it easily could be compared with single error metrics and not only other RMSE4D aggregations of a metric. Additionally extending Crayon to enable fair comparisons of multivariate forecasters would be interesting future work as these introduce additional complexity to the benchmarking system. While improvements can be made, it is important to note that the research questions of this thesis were successfully answered as methods for fair, accurate and reproducible comparisons were identifed and used for a large scale empirical comparison of forecasting methods.

Bibliography

[1] Forecast package in r, https://pkg.robjhyndman.com/forecast/

[2] Forecasting competition for artificial neural networks and computational intelligence, http://www.neural-forecasting-competition.com/NN5/

[3] GluonTS - github repository, https://github.com/awslabs/gluon-ts

[4] GluonTS - probabilistic time series modeling, https://ts.gluon.ai/

[5] Mlbench: Distributed machine learning benchmark, https://mlbench.readthedocs.io/en/latest/

[6] Alexandrov, A., Benidis, K., Bohlke-Schneider, M., Flunkert, V., Gasthaus, J., Januschowski, T., Maddix, D.C., Rangapuram, S., Salinas, D., Schulz, J., Stella, L., Türkmen, A.C., Wang, Y.: GluonTS: Probabilistic time series models in python http://arxiv.org/abs/1906.05264

[7] Baker, M.: 1,500 scientists lift the lid on reproducibility. Nature News 533(7604), 452 (2016)

[8] Bauer, A., Züfle, M., Eismann, S., Grohmann, J., Herbst, N., Kounev, S.: Libra: A benchmark for time series forecasting methods. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering. pp. 189–200 (2021)

[9] Beam, A.L., Manrai, A.K., Ghassemi, M.: Challenges to the reproducibility of machine learning models in health care. Jama 323(4), 305–306 (2020)

[10] Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks 5(2), 157–166 (1994)

[11] Bouthillier, X., Delaunay, P., Bronzi, M., Trofimov, A., Nichyporuk, B., Szeto, J., Mohammadi Sepahvand, N., Raff, E., Madan, K., Voleti, V., et al.: Accounting for variance in machine learning benchmarks. Proceedings of Machine Learning and Systems 3 (2021)

[12] Bühlmann, P., Yu, B.: Analyzing bagging. The annals of Statistics 30(4), 927–961 (2002)

[13] Dau, H.A., Bagnall, A., Kamgar, K., Yeh, C.C.M., Zhu, Y., Gharghabi, S., Ratanamahatana, C.A., Keogh, E.: The ucr time series archive. IEEE/CAA Journal of Automatica Sinica 6(6), 1293–1305 (2019)

[14] Diebold, F.X.: Comparing predictive accuracy, twenty years later: A personal perspective on the use and abuse of diebold–mariano tests. Journal of Business & Economic Statistics 33(1), 1–1 (2015)

[15] Feurer, M., Hutter, F.: Hyperparameter optimization. In: Automated machine learning, pp. 3–33. Springer, Cham (2019)

[16] Freccero, L.A.: Fix dockerfiles in gluonts, `https://github.com/awslabs/gluon-ts/pull/1376`

[17] Freccero, L.A.: Master thesis docker image repository on dockerhub, `https://hub.docker.com/r/arangatang/masterthesis`

[18] Freccero, L.A.: Masterthesis repository on github, `https://github.com/arangatang/masterthesis`

[19] Freccero, L.A.: The runtool, `https://github.com/awslabs/gluon-ts-tools/tree/main/runtool`

[20] GluonTS: Fix negative binomial's scaling, `https://github.com/awslabs/gluon-ts/commit/41bd8a5b869c3f90e0879edb2aacd563c8e3039f`

[21] GluonTS: Fix negative binomial's scaling, `https://github.com/awslabs/gluon-ts/issues/906`

[22] GluonTS: Fix negative binomial's scaling, `https://github.com/awslabs/gluon-ts/pull/909`

[23] Goodwin, P., Lawton, R.: On the asymmetry of the symmetric MAPE 15(4), 405–408, `https://linkinghub.elsevier.com/retrieve/pii/S0169207099000072`

[24] Graves, A.: Generating sequences with recurrent neural networks. CoRR abs/1308.0850 (2013), `http://arxiv.org/abs/1308.0850`

[25] Hassani, H., Silva, E.S.: A kolmogorov-smirnov based test for comparing the predictive accuracy of two sets of forecasts. Econometrics 3(3), 590–609 (2015)

[26] Huang, X., Fox, G.C., Serebryakov, S., Mohan, A., Morkisz, P., Dutta, D.: Benchmarking deep learning for time series: Challenges and directions. In: 2019 IEEE International Conference on Big Data (Big Data). pp. 5679–5682. IEEE, https://ieeexplore.ieee.org/document/9005496/

[27] Hyndman, R.J.: Measuring forecast accuracy p. 9

[28] Hyndman, R.J., Athanasopoulos, G.: Forecasting: principles and practice. OTexts, 3nd edition edn., https://otexts.com/fpp3/

[29] Hyndman, R.J., Billah, B.: Unmasking the theta method. International Journal of Forecasting 19(2), 287–290 (2003)

[30] Ioannidis, J.P., Cripps, S., Tanner, M.A.: Forecasting for covid-19 has failed. International Journal of Forecasting (2020), https://www.sciencedirect.com/science/article/pii/S0169207020301199

[31] jsirol, G.U.: Deepar + negativebinomialoutput performance degradation after upgrading from v0.4.2 to 0.5.0, https://github.com/awslabs/gluon-ts/issues/906

[32] Kim, T.K.: T test as a parametric statistic. Korean journal of anesthesiology 68(6), 540 (2015)

[33] Lai, G., Chang, W.C., Yang, Y., Liu, H.: Modeling long- and short-term temporal patterns with deep neural networks http://arxiv.org/abs/1703.07015

[34] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., Talwalkar, A.: Hyperband: A novel bandit-based approach to hyperparameter optimization. The Journal of Machine Learning Research 18(1), 6765–6816 (2017)

[35] Makridakis, S., Chatfield, C., Hibon, M., Lawrence, M., Mills, T., Ord, K., Simmons, L.F.: The m2-competition: A real-time judgmentally based forecasting study. International Journal of forecasting 9(1), 5–22 (1993)

[36] Makridakis, S., Hibon, M., Lusk, E., Belhadjali, M.: Confidence intervals: An empirical investigation of the series in the m-competition. International Journal of Forecasting 3(3-4), 489–508 (1987)

[37] Makridakis, S., Hibon, M.: The m3-competition: results, conclusions and implications. International Journal of Forecasting 16(4), 451–476 (2000), https://www.sciencedirect.com/science/article/pii/S0169207000000571, the M3- Competition

[38] Makridakis, S., Spiliotis, E., Assimakopoulos, V.: The m4 competition: 100,000 time series and 61 forecasting methods 36(1), 54–74, https://linkinghub.elsevier.com/retrieve/pii/S0169207019301128

[39] Makridakis, S., Spiliotis, E., Assimakopoulos, V., Chen, Z., Gaba, A., Tsetlin, I., Winkler, R.: The m5 uncertainty competition: Results, findings and conclusions (11 2020)

[40] Massey Jr, F.J.: The kolmogorov-smirnov test for goodness of fit. Journal of the American statistical Association 46(253), 68–78 (1951)

[41] Mattson, P., Cheng, C., Coleman, C., Diamos, G., Micikevicius, P., Patterson, D., Tang, H., Wei, G.Y., Bailis, P., Bittorf, V., Brooks, D., Chen, D., Dutta, D., Gupta, U., Hazelwood, K., Hock, A., Huang, X., Ike, A., Jia, B., Kang, D., Kanter, D., Kumar, N., Liao, J., Ma, G., Narayanan, D., Oguntebi, T., Pekhimenko, G., Pentecost, L., Reddi, V.J., Robie, T., John, T.S., Tabaru, T., Wu, C.J., Xu, L., Yamazaki, M., Young, C., Zaharia, M.: MLPerf training benchmark http://arxiv.org/abs/1910.01500

[42] McDermott, M., Wang, S., Marinsek, N., Ranganath, R., Ghassemi, M., Foschini, L.: Reproducibility in machine learning for health. arXiv preprint arXiv:1907.01463 (2019)

[43] Oord, A.v.d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., Kavukcuoglu, K.: WaveNet: A generative model for raw audio http://arxiv.org/abs/1609.03499

[44] Oreshkin, B.N., Carpov, D., Chapados, N., Bengio, Y.: N-BEATS: Neural basis expansion analysis for interpretable time series forecasting http://arxiv.org/abs/1905.10437

[45] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12, 2825–2830 (2011)

[46] Peña, R., Medina, A., Anaya-Lara, O., McDonald, J.R.: Capacity estimation of a minihydro plant based on time series forecasting. Renewable Energy 34(5), 1204–1209 (2009)

[47] Pineau, J., Vincent-Lamarre, P., Sinha, K., Larivière, V., Beygelzimer, A., d'Alché Buc, F., Fox, E., Larochelle, H.: Improving reproducibility in

machine learning research (a report from the neurips 2019 reproducibility program). arXiv preprint arXiv:2003.12206 (2020)

[48] Pritzsche, U.: Benchmarking of classical and machine-learning algorithms (with special emphasis on bagging and boosting approaches) for time series forecasting, https://pdfs.semanticscholar.org/719f/e0e9823ee42630f1c663c102074d86354f67.pdf

[49] Rangapuram, S.S., Seeger, M.W., Gasthaus, J., Stella, L., Wang, Y., Januschowski, T.: Deep state space models for time series forecasting. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31, pp. 7785–7794. Curran Associates, Inc., http://papers.nips.cc/paper/8004-deep-state-space-models-for-time-series-forecasting.pdf

[50] Roelofs, R., Fridovich-Keil, S., Miller, J., Shankar, V., Hardt, M., Recht, B., Schmidt, L.: A meta-analysis of overfitting in machine learning. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems. pp. 9179–9189 (2019)

[51] Sakai, T.: Two sample t-tests for ir evaluation: Student or welch? In: Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval. p. 1045–1048. SIGIR '16, Association for Computing Machinery, New York, NY, USA (2016), https://doi.org/10.1145/2911451.2914684

[52] Salinas, D., Bohlke-Schneider, M., Callot, L., Medico, R., Gasthaus, J.: High-dimensional multivariate forecasting with low-rank gaussian copula processes http://arxiv.org/abs/1910.03002

[53] Salinas, D., Flunkert, V., Gasthaus, J.: DeepAR: Probabilistic forecasting with autoregressive recurrent networks http://arxiv.org/abs/1704.04110

[54] Seabold, S., Perktold, J.: statsmodels: Econometric and statistical modeling with python. In: 9th Python in Science Conference (2010)

[55] Services, A.W.: Aws identity and access management, https://aws.amazon.com/iam/

[56] Services, A.W.: Create a container with your own algorithms and models, https://docs.aws.amazon.com/sagemaker/latest/dg/docker-containers-create.html

[57] Services, A.W.: Sagemaker, https://aws.amazon.com/sagemaker/

[58] Services, A.W.: Sagemaker, https://github.com/arangatang/Crayon

[59] Services, A.W.: Simple storage service, s3, https://aws.amazon.com/s3/

[60] Sherstinsky, A.: Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network 404, 132306, http://arxiv.org/abs/1808.03314

[61] Sivaprasad, P.T., Mai, F., Vogels, T., Jaggi, M., Fleuret, F.: Optimizer benchmarking needs to account for hyperparameter tuning. In: International Conference on Machine Learning. pp. 9036–9045. PMLR (2020)

[62] Smyl, S.: A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting 36(1), 75–85, https://linkinghub.elsevier.com/retrieve/pii/S0169207019301153

[63] Snoek, J., Larochelle, H., Adams, R.P.: Practical bayesian optimization of machine learning algorithms. Advances in neural information processing systems 25 (2012)

[64] Spiliotis, E., Kouloumos, A., Assimakopoulos, V., Makridakis, S.: Are forecasting competitions data representative of the reality? International Journal of Forecasting (12 2018)

[65] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: A simple way to prevent neural networks from overfitting 15(1), 1929–1958, publisher: JMLR.org

[66] Taylor, S.J., Letham, B.: Forecasting at scale, https://peerj.com/preprints/3190v2

[67] Vassilieva, N., Serebryakov, S.: Deep learning benchmarking suite, https://hewlettpackard.github.io/dlcookbook-dlbs

[68] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need p. 11, https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

[69] Wang, Y., Smola, A., Maddix, D.C., Gasthaus, J., Foster, D., Januschowski, T.: Deep factors for forecasting http://arxiv.org/abs/1905.12417

[70] Wen, R., Torkkola, K., Narayanaswamy, B., Madeka, D.: A multi-horizon quantile recurrent forecaster http://arxiv.org/abs/1711.11053

[71] Willmott, C., Matsuura, K.: Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance 30, 79–82, http://www.int-res.com/abstracts/cr/v30/n1/p79-82/

[72] Zhuang, D., Zhang, X., Song, S.L., Hooker, S.: Randomness in neural network training: Characterizing the impact of tooling. arXiv preprint arXiv:2106.11872 (2021)