GluonTS: Probabilistic Time Series Models in Python

Alexander Alexandrov, Konstantinos Benidis, Michael Bohlke-Schneider, Valentin Flunkert, Jan Gasthaus, Tim Januschowski, Danielle C. Maddix, Syama Rangapuram, David Salinas, Jasper Schulz, Lorenzo Stella, Ali Caner Türkmen, Yuyang Wang
Amazon Web Services

Editor:

Abstract

We introduce Gluon Time Series (GluonTS)[1], a library for deep-learning-based time series modeling. GluonTS simplifies the development of and experimentation with time series models for common tasks such as forecasting or anomaly detection. It provides all necessary components and tools that scientists need for quickly building new models, for efficiently running and analyzing experiments and for evaluating model accuracy.

1. Introduction

Large collections of time series are ubiquitous and occur in areas as different as natural and social sciences, internet of things applications, cloud computing, supply chains and many more. These datasets have substantial value, since they can be leveraged to make better forecasts or to detect anomalies more effectively, which in turn results in improved downstream decision making. Traditionally, time series modeling has focused (mostly) on individual time series via *local* models.[2] In recent years, advances in deep learning have led to substantial improvements over the local approach by utilizing the large amounts of data available for estimating parameters of a single *global* model over the entire collection of time series. For instance, recent publications (Flunkert et al., to appear; Wen et al., 2017; Laptev et al., 2017) and winning models of forecasting competitions (Makridakis et al., 2018; Smyl et al., 2018) have shown significant accuracy improvements via the usage of deep learning models trained jointly on large collections of time series.

Despite the practical importance of time series models, evidence that deep learning based methods lead to improved models and the success of deep-learning-based toolkits in other domains (Hieber et al., 2018; Dai et al., 2018; Bingham et al., 2018), there exists, to the best of our knowledge, currently no such toolkit for time series modeling.

We fill this gap with GluonTS (`https://gluon-ts.mxnet.io`) – a deep learning library that bundles components, models and tools for time series applications such as forecasting or anomaly detection. GluonTS simplifies all aspects of scientific experiments with time series models. It includes components such as distributions, neural network architectures for sequences, and feature processing steps which can be used to quickly assemble and train new

1. `https://gluon-ts.mxnet.io`
2. In local models, the free parameters of the time series model are estimated per individual time series in the collection of time series.

models. Apart from supporting pure deep-learning-based models, GluonTS also includes probabilistic models and components such as state-space models and Gaussian Processes. This allows scientists to combine the two approaches, which is a promising current research direction (e.g., (Rangapuram et al., 2018; Krishnan et al., 2017; Fraccaro et al., 2016)). The library also provides tools rapid experimentation including convenience functions for data I/O, model evaluation and plotting utilities. The library is based on the Apache MXNet (Chen et al., 2015) deep learning framework and more specifically on the *Gluon* API[3].

GluonTS's main use-case is in building new time series models. To show-case its model building capabilities and facilitate benchmarking, it comes with implementations for a number of already published models. In this paper, we provide a benchmark of pre-bundled models in GluonTS on a set of public datasets.

We target GluonTS mainly as a toolkit for scientists who are working with time series datasets and wish to experiment with new models or solve specific problems. Accordingly, we have designed GluonTS such that it scales from small to large datasets as this is the primary field of activity of scientists.

This paper is structured as follows. In Sec. 2 we discuss the general design principles and architecture of the library, and discuss the different components available in GluonTS. In Sec. 3, we formally introduce a number of time series problems which GluonTS allows to address. Sec. 4 provides an overview of common neural forecasting architectures that can be assembled with GluonTS or are implemented as a pre-bundled benchmark model. In Sec. 5, we run our benchmark models of published deep learning based forecasting models on 11 public datasets and demonstrate the applicability of these models to other tasks such as anomaly detection. We discuss related work in Sec. 6 and conclude in Sec. 7 with pointers to future work.

2. Library design and components

In the design and structure of GluonTS, we follow these principles.

Modularity: Components are decoupled from each other and are designed with clear interfaces. This allows users to build models by combining and/or extending components in new ways.

Scalability: All components scale from running on a few time series to large datasets. For instance, data processing relies on streaming, so that the datasets do not need to be loaded into memory.

Reproducibility: To make experiments easily reproducible, components serialize in a human readable way. This allows to "log" configured models and experiment setups. The user can then later fully reproduce or inspect the experimental setup from the log (Appendix B contains details.)

Listing 1 shows a simple workflow for creating and training a pre-built forecasting model, and evaluating the model in a backtest. We describe the main components of the workflow before describing how to assemble new models.

3. https://mxnet.apache.org/gluon

```python
from swist.dataset.repo import DatasetRepository
from swist.dataset.util import to_pandas
from swist.model.deepar import DeepAREstimator
from swist.trainer import Trainer
from swist.evaluation import Evaluator
from swist.evaluation.backtest import backtest_metrics

repo = DatasetRepository(allow_download=True)
meta, train_ds, test_ds = repo.get('electricity')

estimator = DeepAREstimator(
  freq=meta.freq,
  prediction_length=100,
  batch_size=32,
  trainer=Trainer(epochs=20)
)
predictor = estimator.train(train_ds)

forecasts = predictor.predict(test_ds)
for forecast, ts in zip(forecasts, test_ds):
  to_pandas(ts).plot()
  forecast.plot()

evaluator = Evaluator(quantiles=(0.1, 0.5, 0.9))
agg_metrics, item_metrics = backtest_metrics(
        train_dataset=train_ds,
        test_dataset=test_ds,
        estimator=predictor,
        evaluator=evaluator
)
```

Listing 1: Model training and evaluation in GluonTS

2.1 Data I/O and processing

GluonTS has two types of data sources that allow a user to experiment and benchmark algorithms. The first is a `DatasetRepository` that contains a number of public time series datasets, and is extensible with custom private datasets. These input dataset can be included in jsonlines or parquet format.

The second data source is a generator for synthetic time series. The user can configure the generator in a declarative way to generate complex time series that exhibit e.g. seasonality, trend and different types of noise. Along with the time series itself, the generator can produce different types of co-variates, such as categorical co-variates, and allow the properties of the generated time series to depend on these.

From a high level perspective, data handling in GluonTS is done on streams (Python iterators) of dictionaries. During feature processing, a `DatasetReader` loops over the input dataset, and emits a stream of dictionaries. The feature processing pipelines consists of a sequence of `Transformation`s that act successively on the stream of dictionaries. Each `Transformation` can add new keys or remove keys, modify values, filter items from the stream or add new items.

GluonTS contains a set of time series specific transformations that include splitting and padding of time series (e.g. for evaluation splits), common time series transformation such as Box-Cox transformations or marking of special points in time and missing values.A user can easily include custom transformations for specific purposes, and combine them with existing transformations in a pipeline.

2.2 Predictor

GluonTS follows a stateless predictor API as is e.g. used in scikit-learn (Pedregosa et al., 2011) and SparkML (Meng et al., 2015). An `Estimator` has a `train` method that takes a (train) dataset and returns a `Predictor`. The `Predictor` can then be invoked on a (test) dataset or a single item to generate predictions. Classical forecasting models that estimate parameters per time series before generating predictions are pure predictors in this API. GluonTS faciliates comparisons with established forecasting packages (e.g. (Hyndman and Khandakar, 2008; Taylor and Letham, 2017)).

2.3 Distribution / Output

GluonTS contains a flexible abstraction for probability distributions (and densities), which are common building blocks in probabilistic time series modeling. Concrete implementations include common parametric distributions, such as Gaussian, Student's t, gamma, and negative binomial. There is also a binned distribution that can be used to quantize a signal. The binned distribution is configured with bin edges and bin values that represent the bins, e.g. when samples are drawn.

A `TransformedDistribution` allows the user to map a distribution using a differentiable bijection (Dillon et al., 2017; Rezende and Mohamed, 2015). A bijection can be a simple fixed transformation, such as a log-transform that maps values into a different domain. Transformations can also be arbitrarily complex (as long as they are invertible, and have a tractable Jacobian determinant), and can depend on learnable parameters that will be estimated jointly with the other model parameters. In particular, this paradigm covers the popular Box-Cox transformations (Box and Cox, 1964). GluonTS also supports mixtures of arbitrary base distributions.

2.4 Forecast object

Trained forecast models (i.e. predictors) return `Forecast` objects as predictions, which contain a time index (start, end and time granularity) and a representation of the probability distribution of values over the time index. Different models may use different techniques for estimating and representing this joint probility distribution.

The auto-regressive models (Section 4.3) naturally generate sampled trajectories, and in this case, the joint distribution is represented through a collection of (e.g. $n = 1000$) sample

paths – potential future times series trajectories. Any desired statistic can then be extracted from this set of sample paths. For instance, in retail, a distribution of the total number of sales in a time interval can be estimated, by summing each individual sample path over the interval. The result is a set of samples from the distribution of total sales (Seeger et al., 2016; Flunkert et al., to appear).

Other models output forecasts in the form of quantile predictions for a set of quantiles and time intervals (Wen et al., 2017; Gasthaus et al., 2019). The corresponding `QuantileForecast` then stores this collection of quantile values and time intervals.

Forecast objects in GluonTS have a common interface that allows the `Evaluation` component to compute accuracy metrics. In particular, each forecast object has a `.quantile` method that takes a desired p-value (e.g. $p = 0.9$) and an optional time index, and returns the estimate of the corresponding quantile value in that time index.

2.5 Evaluation

For quantitatively assessing the accuracy of its time series models, GluonTS has an `Evaluator` class that can be invoked on a stream of `Forecast` objects and a stream of true targets (pandas data frame).

The `Evaluator` iterates over true targets and forecasts in a streaming fashion, and calculates metrics, such as quantile loss, Mean Absolute Scale Error (MASE), Mean Absolute Percent Error (MAPE) and Scaled Mean Absolute Percent Error (sMAPE). The `Evaluator` return these per item metrics as a pandas data frame. This makes it easy for a user to explore or visualize statistics such as the error distribution across a data set. Figure 1 shows an example histogram of MASE values for the ETS, Prophet and DeepAR method (see Sec. 4 for details on these models) on the same dataset. In addition to the per item metrics, the `Evaluator` also calculates aggregate statistics over the entire dataset, such as Weighted Mean Absolute Percent Error (wMAPE) or weighted quantile loss, which are useful for instance for automatic model tuning e.g. using SageMaker.

For qualitatively assessing the accuracy of time series models, GluonTS contains methods that visualize time series and forecasts using matplotlib (Hunter, 2007).

In addition to the `Evaluator`, GluonTS allows the user to run full backtest evaluations. The `backtest_metrics` method runs a test scenario that trains an estimator on a training dataset, or uses a pre-trained predictor, and evaluates the model on a test dataset. A separate `Splitter` component generates train/validation/test splits from an existing dataset. These splits can be simple single time point splits or more complex test sets supporting for instance evaluation over multiple different time points or rolling windows. A user can easily combine these components to run simple or complex training and evaluation scenarios in a reproducible manner.

2.6 Time Series Model Building

GluonTS's primary purpose is the creation of new models. A user can implement a new model fully from scratch by just adhering to the `Estimator` / `Predictor` API. In most cases however, it is simpler to use a pre-structured model template. For instance, to create a new deep learning based forecast model, one can use the `GluonForecastEstimator` and implement and configure the following members and methods:

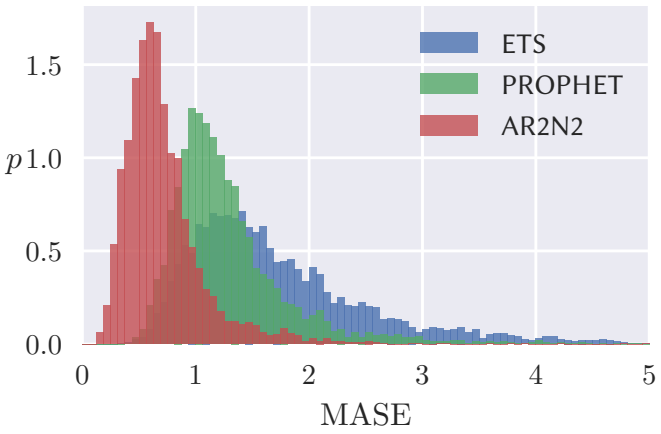

Figure 1: Histogram of forecast error measured in MASE on traffic-dataset for ETS, Prophet and DeepAR.

- `train_network_cls`: A neural network that returns the training loss for the model and will be called during training.
- `pred_network_cls`: A neural network that uses the same parameters as the training network, but instead of returning a loss returns the forecast samples.
- `create_transformation`: A method that returns the transformation pipeline.[4]
- `forecast_cls`: The forecast class that will be wrapped around the output of the forecast network.

Appendix C presents an example containing a simple model that estimates the median for each future time point independently.We describe more complex models in Sec. 4 which can be implemented readily in GluonTS with the abstractions shown above.

3. Time Series Problems

GluonTS addresses probabilistic modeling of uni- or multi-variate sequences of (large) collections of time series. Important applications include forecasting, smoothing and anomaly detection.

More formally, let $Z = \{z_{i,1:T_i}\}_{i=1}^{N}$ be a set of N univariate time series, where $z_{i,1:T_i} := (z_{i,1}, z_{i,2}, \ldots, z_{i,T_i})$, and $z_{i,t} \in \mathbb{R}$ denotes the value of the i-th time series at time t. We mainly consider time series where the time points are equally spaced (Sec. **??** is an exception) but the time units across different sets can be arbitrary (e.g. hours, days, months). Further, the time series do not have to be aligned, i.e., the starting point $t = 1$ can refer to a different absolute time point for different time series i.

Furthermore, let $X = \{\mathbf{x}_{i,1:T_i+\tau}\}_{i=1}^{N}$ be a set of associated, time-varying covariate vectors with $\mathbf{x}_{i,t} \in \mathbb{R}^D$.

4. Since the transformation pipeline can itself contain learnable parameters and the `Estimator` is stateless this is a method returning the transformation rather than a fixed member.

The goal of *forecasting* (Hyndman and Athanasopoulos, 2017) is to predict the probability distribution of future values $z_{i,T_{i+1}:T_{i+\tau}}$ given the past values $z_{i,1:T_i}$, the covariates $\mathbf{x}_{i,1:T_{i+\tau}}$, and the model parameters Φ:

$$p\left(z_{i,T_i+1:T_i+\tau} \mid z_{i,1:T_i}, \mathbf{x}_{i,1:T_i+\tau}; \Phi\right). \tag{1}$$

Smoothing or *missing value imputation* in time series can leverage the sequence structure, and therefore allow for more sophisticated approaches compared to the general missing value imputation setting (e.g., (Biessmann et al., 2018)). Missing values are commonly found in real-world time series collections. For instance, in retail demand forecasting, the true demand is not observed when the item is not in-stock, i.e., cannot be sold. Smoothing is similar to forecasting, except that the time points that we want to predict do not lie in the future. Instead, for a set of series and *arbitrary* time points there are missing or unobserved values, and the goal is to estimate the (joint) probability distribution over these missing values, i.e.,

$$p\left(z_{i,j_1:j_\tau} \mid z_{i,k_1:k_\tau}, \mathbf{x}_{i,k_1:x_{k\tau}}; \Phi\right).$$

where $\{j_1 : j_\tau\}$ are the indexes of the missing values and $\{k_1 : k_\tau\}$ the indexes of the observed values, such that $\{j_1 : j_\tau\} \cup \{k_1 : k_\tau\} = \{1 : T_i\}$.

In *anomaly* or *outlier detection* we want to identify time points, where the observed values are unlikely to have occurred. While we may have additional labels that annotate anomalous behavior, it is in many applications not feasible to directly train a classifier on these labels, because the labels are too sparse – after all, anomalies are often rare.

In this unsupervised case, anomaly detection is similar to forecasting, except that all values are observed and we want to know how likely they were. A probabilistic forecasting model can be converted into an anomaly detection model in different ways. For univariate models where the cumulative distribution function (CDF) of the marginal predicted distribution can be evaluated, one can directly calculate the p-value (tail probability) of a newly observed data point via the CDF (Shipmon et al., 2017). For other models or in the multivariate case, we can use the log-likelihood values. This is slightly more complicated, since the range of log-likelihood values depends on the model and the dataset, however, we can estimate multiple high percentiles of the distribution of negative log-likelihood values on the training set, such as the 99, 99.9, 99.99 etc. At prediction time, we can evaluate the negative log-likelihood of each observations under the model sequentially, and compare this with the estimated quantiles. Both approaches allow us to mark unlikely points or unlikely sequences of observations as anomalies. The quality of this forecasting based approach depends on the frequency of such anomalies in the training data, and works best if known anomalies are removed or masked. If some labels for anomalous points are available, they can be used to set likelihood thresholds or to evaluate model accuracy e.g. by calculating classification scores, such as precision and recall.

4. Time Series Models

All tasks from Sec. 3 have at their core the estimation of a joint probability distribution over time series values at different time points. We model this as a supervised learning problem, by fixing a model structure upfront and learning the model parameters Φ using a statistical

optimization method, such as maximum likelihood estimation, and the sets Z and X as training data.

Classical models that were developed for these tasks (Hyndman and Athanasopoulos, 2017), are, with some exceptions (Chapados, 2014), local models that learn the parameters Φ for each time series individually. Recently, however, several neural time series models have been proposed (Flunkert et al., to appear; Gasthaus et al., 2019; Rangapuram et al., 2018; Laptev et al., 2017; Wen et al., 2017) where a single global model is learned for all time series in the dataset by sharing the parameters Φ.[5] Due to their capability to extract higher-order features, these neural methods can identify complex patterns within and across time series from datasets consisting of raw time series (Flunkert et al., to appear; Laptev et al., 2017) with considerably less human effort.

Time series models can be broadly categorized as generative and discriminative, depending on how the target Z is modeled (Ng and Jordan, 2002).[6]

CATEGORY	MODELING
generative	$p(z_{i,1:T_i+\tau}\|\mathbf{x}_{i,1:T_i+\tau};\Phi)$
discriminative	$p(z_{i,T_i+1:T_i+\tau} \mid z_{i,1:T_i}, \mathbf{x}_{i,1:T_i+\tau};\Phi)$

Generative models assume that the given time series are generated from an unknown stochastic process $p(Z|X;\Phi)$ given the covariates X. The process is typically assumed to have some parametric structure with unknown parameters Φ. Prominent examples include classical models such as ARIMA and ETS (Hyndman et al., 2008), Bayesian structural time series (BSTS) (Scott and Varian, 2014) and the recently proposed deep state space model (Rangapuram et al., 2018). The unknown parameters of this stochastic process are typically estimated by maximizing the likelihood, which is the probability of the observed time series, $\{z_{i,1:T_i}\}$, under the model $p(Z|X;\Phi)$, given the covariates $\{\mathbf{x}_{i,1:T_i}\}$. Once the parameters Φ are learned, the forecast distribution in Eq. (1) can be obtained from $p(Z|X;\Phi)$. In contrast to ETS and ARIMA, which learn Φ per time series individually, neural generative models like (Rangapuram et al., 2018) further express Φ as a function of a neural network whose weights are shared among all time series and learned from the whole training data.

Discriminative models such as (Flunkert et al., to appear; Gasthaus et al., 2019; Wen et al., 2017), model the conditional distribution (for a fixed τ) from Eq. (1) directly via a neural network. Compared to generative models, conditional models are more flexible, since they make less structural assumptions, and hence are also applicable to a broader class of application domains. In the following, we distinguish between auto-regressive and sequence-to-sequence models among discriminative models. Further distinctions within the subspace of discriminative models are discussed in more detail in (Faloutsos et al., 2018).

5. While neural network based forecasting methods have been used for a long time, they have been traditionally employed as local models. This led to mixed results compared with other local modeling approaches (Zhang et al., 1998).
6. Note that our categorization overloads the distinction used in Machine Learning. Technically, neither of the category defined here jointly models the covariates X and the target Z and thus both belong to "discriminative" models in the traditional sense.

4.1 Generative Models

Here we describe in detail the forecasting methods implemented in GluonTS that fall under the generative models category. Further examples in this family include Gaussian Processes (Girard et al., 2003) and Deep Factor models (Maddix et al., 2018; Wang et al., 2019) which we omit due to space restrictions. **State Space Models** (SSMs) provide a principled framework for modeling and learning time series patterns (Hyndman et al., 2008; Durbin and Koopman, 2012; Seeger et al., 2016). In particular, SSMs model the temporal structure of the data via a latent state $\boldsymbol{l}_t \in \mathbb{R}^D$ that can be used to encode time series components, such as level, trend, and seasonality patterns. A general SSM is described by the so-called state-transition equation, defining the stochastic transition dynamics $p(\boldsymbol{l}_t|\boldsymbol{l}_{t-1})$ by which the latent state evolves over time, and an observation model specifying the conditional probability $p(z_t|\boldsymbol{l}_t)$ of observations given the latent state. Several classical methods (e.g., ETS, ARIMA) can be cast under this framework by choosing appropriate transition dynamics and observation model (Hyndman et al., 2008).

A widely used special case is the linear innovation state space model, where the transition dynamics and the observation model are given by. Note that we drop the index i from the notation here since these models are typically applied to individual time series.

$$\begin{aligned} \boldsymbol{l}_t &= \boldsymbol{F}_t \boldsymbol{l}_{t-1} + \boldsymbol{g}_t \varepsilon_t, & \varepsilon_t &\sim \mathcal{N}(0,1), \\ z_t &= y_t + \sigma_t \epsilon_t, \quad y_t = \boldsymbol{a}_t^\top \boldsymbol{l}_{t-1} + b_t, & \epsilon_t &\sim \mathcal{N}(0,1). \end{aligned}$$

Here $\boldsymbol{a}_t \in \mathbb{R}^L$, $\sigma_t \in \mathbb{R}_{>0}$ and $b_t \in \mathbb{R}$ are further (time-varying) parameters of the model. Finally, the initial state $\boldsymbol{l}_0$ is assumed to follow an isotropic Gaussian distribution, $\boldsymbol{l}_0 \sim N(\boldsymbol{\mu}_0, \mathrm{diag}(\boldsymbol{\sigma}_0^2))$.

The state space model is fully specified by the parameters $\Theta_t = (\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0, \boldsymbol{F}_t, \boldsymbol{g}_t, \boldsymbol{a}_t, b_t, \sigma_t)$, $\forall t > 0$. One generic way of estimating them is by maximizing the marginal likelihood, i.e.,

$$\Theta^*_{1:T} = \operatorname*{argmax}_{\Theta_{1:T}} p_{SS}(z_{1:T}|\Theta_{1:T}),$$

where $p_{SS}(z_{1:T}|\Theta_{1:T})$ denotes the marginal probability of the observations $z_{1:T}$ given the parameters Θ under the state space model, integrating out the latent state $\boldsymbol{l}_t$.

Deep State Space Models (Rangapuram et al., 2018) (referred as DeepState here) is a probabilistic time series forecasting approach that combines state space models with deep learning. The main idea is to parametrize the linear SSM using a recurrent neural network (RNN) whose weights are learned jointly from a dataset of raw time series and associated covariates. More precisely, DeepState learns a globally shared mapping $\Psi(\mathbf{x}_{i,1:t}, \Phi)$ from the covariate vectors $\mathbf{x}_{i,1:T_i}$ associated with each target time series $z_{i,1:T_i}$, to the (time-varying) parameters $\Theta_{i,t}$ of a linear state space model for the i-th time series. The mapping Ψ from covariates to state space model parameters is parametrized using a deep recurrent neural network (RNN). Given the covariates[7] $\mathbf{x}_{i,t}$ associated with time series $z_{i,t}$, a multi-layer recurrent neural network with LSTM cells and parameters Φ, it computes a representation of the features via a recurrent function h,

$$\mathbf{h}_{i,t} = h(\mathbf{h}_{i,t-1}, \mathbf{x}_{i,t}, \Phi).$$

7. The covariates (features) can be time dependent (e.g. product price or a set of dummy variables indicating day-of-week) or time independent (e.g., product brand, category etc.).

The real-valued output vector of the last LSTM layer is then mapped to the parameters $\Theta_{i,t}$ of the state space model, by applying affine mappings followed by suitable elementwise transformations constraining the parameters to appropriate ranges.

Then, given the features $\mathbf{x}_{i,1:T}$ and the parameters Φ, under this model, the data $z_{i,1:T_i}$ is distributed according to

$$p(z_{i,1:T_i}|\mathbf{x}_{i,1:T_i}, \Phi) = p_{SS}(z_{i,1:T_i}|\Theta_{i,1:T_i}), \qquad i = 1, \ldots, N. \tag{2}$$

where p_{SS} denotes the marginal likelihood under a linear state space model, given its (time-varying) parameters $\Theta_{i,t}$. Parameters $\Theta_{i,t}$ are then used to compute the likelihood of the given observations $z_{i,t}$, which is used for learning of the network parameters Φ.

4.2 Discriminative Models

Inspired by the sequence-to-sequence learning approach presented in (Sutskever et al., 2014), several forecasting methods are proposed in this framework (Wen et al., 2017). These sequence-to-sequence models consist of a so-called encoder network that reads in a certain context of the training range of the time series (i.e., $z_{i,1:T_i}, \mathbf{x}_{i,1:T_i}$), and encodes information about the sequence in a latent state. This information is then passed to the decoder network, which generates the forecast $z_{i,T_i+1:T_i+\tau}$ by combining the latent information with the features $\mathbf{x}_{i,T_i+1:T_i+\tau}$ in the prediction range.

While sequence-to-sequence models are more complex and may need more training data compared to simple auto-regressive models, they have several advantages. First, they can handle differences in the set of features available in the training and prediction ranges. This enables the use of time series covariates that are not available in the future, which is a typical problem often encountered in practical applications. Second, the decoder does not have to be an auto-regressive architecture. During the multi-step ahead prediction, the prediction from the previous step does not need to be used as a feature for the current time step. This avoids error accumulation, which can be beneficial especially for long forecast horizons.

Compared to auto-regressive models, which represent the sequential nature of the time series well, a sequence-to-sequence approach is more akin to multivariate regression. This means that the prediction horizon τ has to be fixed beforehand in sequence-to-sequence models, and a complete retraining is needed if the forecast is required beyond τ steps.

GluonTS contains a flexible sequence-to-sequence framework that makes it possible to combine generic encoder and decoder networks to create custom sequence-to-sequence models. Moreover, GluonTS also has example implementations of specific forecasting models (Wen et al., 2017) that fall under this category as well as generic models (Vaswani et al., 2017) that have been proven to be successful in this domain.

Neural Quantile Regression Models. Quantile regression (Koenker, 2005) is a technique for directly predicting a particular quantile of a dependent variable. These techniques have been combined with deep learning and employed in the context of time series forecasting (Xu et al., 2016; Wen et al., 2017). More specifically, within the sequence-to-sequence framework, we can employ a decoder that directly outputs a set of quantile values for each time step in the forecast horizon, and train such a model using the quantile loss function.

We have implemented variants of such quantile decoder models in GluonTS following (Wen et al., 2017) and combined them with RNN and Dilated Causal Convolution (CNN) encoders, resulting in models dubbed RNN-QR and CNN-QR, respectively.

Transformer. GluonTS also contains an implementation of the Transformer architecture (Vaswani et al., 2017) that has been successful in natural language processing. The Transformer model captures the dependencies of a sequence by relying entirely on attention mechanisms. The elimination of the sequential computation makes the representation of each time step independent of all other time steps and therefore allows the parallelization of the computation.

In a nutshell, the Transformer architecture uses stacked self-attention and point-wise, fully connected layers for both the encoder and the decoder, while the decoder has an additional cross-attention layer that is applied on the encoder output. In our implementation we use the same architecture as in (Vaswani et al., 2017). However, since (Vaswani et al., 2017) focuses on the natural language processing problem, we replace the softmax distribution over discrete values (e.g. characters or words) by the generic distribution component discussed in Sec. 2.3 so our model fits the probabilistic time series forecasting framework.

4.3 Auto-regressive models

Auto-regressive models reduce the sequence prediction task fully to a one-step-ahead problem where we model

$$p\left(z_{i,T_i+1} \middle|\ z_{i,1:T_i}, \mathbf{x}_{i,1:T_i+\tau}; \Phi\right).$$

The model is trained on a sequence by sequential one-step-ahead predictions, and the error is aggregated over the sequence for the model update. For prediction, the model is propagated forward in time by feeding in samples, Through multiple simulations, a set of sample-paths representing the joint probability distribution over the future of the sequence is obtained.

NPTS. The Non-Parametric Time Series forecaster (NPTS) (Gasthaus, 2016) falls into the class of simple forecasters that use one of the past observed targets as the forecast for the current time step. Unlike the naive or seasonal naive forecasters that use a fixed time index (the previous index $T-1$ or the past season $T-\tau$) as the prediction for the time step T, NPTS randomly samples a time index $t \in \{0, \dots, T-1\}$ in the past to generate a prediction sample for the current time step T. By sampling multiple times, one obtains a Monte Carlo sample from the predictive distribution, which can be used e.g. to compute prediction intervals. More precisely, given a time series $z_0, z_1, \dots, z_{T-1}$, the generative process of NPTS for time step T is given by

$$\hat{z}_T = z_t, \quad t \sim q_T(t), \quad t \in \{0, \dots, T-1\},$$

where $q_T(t)$ is categorical probability distribution over the indices in the training range. Note that the expectation of the prediction, $\mathbb{E}[\hat{z}_T]$ is given by

$$\mathbb{E}[\hat{z}_T] = \sum_{t=0}^{T-1} q_T(t) z_t,$$

i.e. in expectation the model reduces to an autoregressive (AR) model with weights $q_T(\cdot)$, and can thus be seen as a non-parametric probabilistic extension of the AR framework.

NPTS uses the following sampling distribution q_T, which uses weights that decay exponentially according to the distance of the past observations,

$$q_T(t) = \frac{\exp(-\alpha\,|T-t|)}{\sum_{t'=0}^{T-1}\exp(-\alpha\,|T-t|)}, \tag{3}$$

where α is a hyper-parameter that is adjusted based on the data. In this way, the observations from the recent past are sampled with much higher probability than the observations from the distant past.

The special case, where one uses uniform weights for all the time steps in the training range, i.e., $q_t = 1$, leads to the Climatological forecaster. This can equivalently be achieved by letting $\alpha \to 0$ in Eq. 3. Similarly, by letting $\alpha \to \infty$, we recover the naive forecaster, which always predicts the last observed value. One strength of NPTS as a baseline method is that it can be used to smoothly interpolate between these two extremes by varying α.

So far we have only described the one step ahead forecast. To generate forecasts for multiple time steps, one can simply absorb the predictions for the last time steps into the observed targets, and then generate subsequent predictions using the last T targets. For example, prediction for time step $T + t, t > 0$ is generated using the targets $z_t, \ldots, z_{T-1}, \hat{z}_T, \ldots, \hat{z}_{T+t-1}$.

DeepAR. GluonTS contains an auto-regressive RNN time series model, DeepAR, which is similar to the architectures described in (Flunkert et al., to appear; Gasthaus et al., 2019). DeepAR consists of a RNN (either using LSTM or GRU cells) that takes the previous time points and co-variates as input. DeepAR then either estimates parameters of a parametric distribution (see Sec. 2.3) or a highly flexible parameterization of the quantile function. Training and prediction follow the general approach for auto-regressive models discussed above.

In contrast to the original model description in (Flunkert et al., to appear), DeepAR not only takes the last target value as an input, but also a number of lagged values that are relevant for the given frequency. For instance, for hourly data, the lags may be 1 (previous time point), 1×24 (one day ago), 2×24 (two days ago), 7×24 (one week ago) etc.

Wavenet (van den Oord et al., 2016) is an auto-regressive neural network with dilated causal convolutions at its core. In the set of GluonTS models, it represents the archetypical auto-regressive Convolutional Neural Network (CNN) models. While it was developed for speech synthesis tasks, it is in essence a time series model that can be used for time series modeling in other problem domains. In the text-to-speech application, the output is a bounded signal and in many implementations the value range is often quantized into discrete bins. This discretized signal is then modeled using a flexible softmax distribution that can represent arbitrary distributions over the discrete values, at the cost of discarding ordinal information.

As discussed in Sec. 2.3 GluonTS supports such discretized distributions via a binned distribution. The neural architecture of Wavenet is not tied to a quantized output, and can be readily combined with any of the distributions available in GluonTS.

estimator dataset	**Auto-ARIMA**	**Auto-ETS**	**Prophet**	**NPTS**	**Transformer**	**CNN-QR**	**DeepAR**
SP500-returns	0.975 +/- 0.001	0.982 +/- 0.001	0.985 +/- 0.001	**0.832 +/- 0.000**	0.836 +/- 0.001	0.907 +/- 0.006	0.837 +/- 0.002
electricity	0.056 +/- 0.000	0.067 +/- 0.000	0.094 +/- 0.000	**0.055 +/- 0.000**	0.062 +/- 0.001	0.081 +/- 0.002	0.065 +/- 0.006
m4-Daily	0.024 +/- 0.000	**0.023 +/- 0.000**	0.090 +/- 0.000	0.145 +/- 0.000	0.028 +/- 0.000	0.026 +/- 0.001	0.028 +/- 0.000
m4-Hourly	0.040 +/- 0.001	0.044 +/- 0.001	0.043 +/- 0.000	0.048 +/- 0.000	0.042 +/- 0.010	0.065 +/- 0.008	**0.034 +/- 0.004**
m4-Monthly	**0.097 +/- 0.000**	0.099 +/- 0.000	0.132 +/- 0.000	0.233 +/- 0.000	0.134 +/- 0.002	0.126 +/- 0.002	0.135 +/- 0.003
m4-Quarterly	0.080 +/- 0.000	**0.078 +/- 0.000**	0.123 +/- 0.000	0.255 +/- 0.000	0.095 +/- 0.003	0.091 +/- 0.000	0.091 +/- 0.001
m4-Weekly	**0.050 +/- 0.000**	0.051 +/- 0.000	0.108 +/- 0.000	0.296 +/- 0.001	0.075 +/- 0.005	0.056 +/- 0.000	0.072 +/- 0.003
m4-Yearly	0.124 +/- 0.000	0.126 +/- 0.000	0.156 +/- 0.000	0.355 +/- 0.000	0.127 +/- 0.004	0.121 +/- 0.000	**0.120 +/- 0.002**
parts	1.403 +/- 0.002	1.342 +/- 0.002	1.637 +/- 0.002	1.355 +/- 0.002	1.000 +/- 0.003	**0.901 +/- 0.000**	0.972 +/- 0.005
traffic	-	0.462 +/- 0.000	0.273 +/- 0.000	0.162 +/- 0.000	0.132 +/- 0.010	0.186 +/- 0.002	**0.127 +/- 0.004**
wiki10k	0.610 +/- 0.001	0.841 +/- 0.001	0.681 +/- 0.000	0.452 +/- 0.000	0.294 +/- 0.008	0.314 +/- 0.002	**0.292 +/- 0.021**

Table 1: CRPS error for all methods. Ten runs are done for each method the Mean and std are computed over 10 runs. Missing value indicates the method did not complete in 24 hours.

5. Experiments

In this section we demonstrate the practical effectiveness of a subset of the forecast and anomaly detection models discussed in Sec. 4.

Table 1 shows the CRPS (Gasthaus et al., 2019) accuracy of different forecast methods in GluonTS on the following 11 public datasets:

- S&P500: daily difference time series of log-return of stocks from S&P500
- electricity: hourly time series of the electricity consumption of 370 customers (Dheeru and Karra Taniskidou, 2017)
- m4: 6 datasets from the M4 competition (Makridakis et al., 2018) with daily, hourly, weekly, monthly, quaterly and yearly frequencies
- parts: monthly demand for car parts used in (Seeger et al., 2016)
- traffic: hourly occupancy rate, between 0 and 1, of 963 car lanes of San Francisco bay area freeways (Dheeru and Karra Taniskidou, 2017)
- wiki10k: daily traffic of 10K Wikipedia pages

The hyperparameters of each methods are kept constant across all datasets and we train with 5000 gradient updates the neural networks models. Each method is run on a 8 cores machine with 16 GB of RAM. The neural network methods compare favorably to established methods such as ARIMA or ETS. The total running time of the methods (training + prediction) is included in Appendix A. Running times are similar for the GluonTS-based models. Most of them can be trained and evaluated under an hour.

Regarding accuracy, there is no overall dominating method. Hence, the experiment illustrates the need for a flexible modeling toolkit, such as GluonTS, that allows to assemble models quickly for the data set and application at hand.

The accuracy of neural forecast methods for time series anomaly detection has been previously demonstrated (Shipmon et al., 2017). Here, we provide exemplary plots for anomalies detected using the forecast models discussed above. Starting from a trained

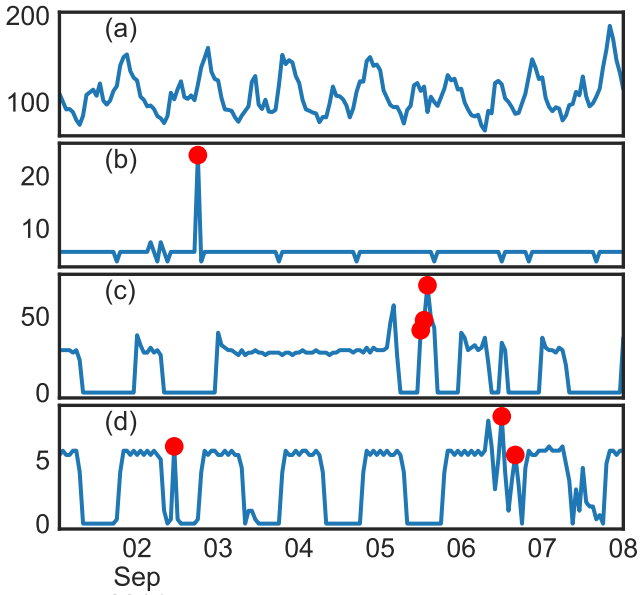

Figure 2: Examples of anomalies detected using a trained DeepAR forecast model on the electricity dataset.

DeepAR forecast model on the electricity dataset, we use the CDF at each time point and mark values as anomalies using a p-value of 10^{-4}. The results are depicted in Fig. 2. The data consists of hourly observations and we plot a detection window of one week. Most of the time series do not exhibit anomalies in this time frame, as shown in panel (a). Panel (b)–(d) depict example time series for which the model detected anomalous behavior. The detected anomalies are points that do not match the series historical behavior such as seasonality or noise level. These plots demonstrate the applicability of the GluonTS neural time series models for anomaly detection. Clearly, further research and experiments are necessary to e.g. quantify the relation between forecast accuracy and detection accuracy.

6. Related Work

Deep learning frameworks, such as (Chen et al., 2015; Paszke et al., 2017; Abadi et al., 2016) are growing in popularity. In recent years, more application-specific toolkits have appeared primarily in those areas where deep learning has been overwhelmingly successful, such as computer vision and language-related tasks, but also increasingly beyond (Hieber et al., 2018; Dai et al., 2018; Bingham et al., 2018). For forecasting, we are not aware of packages based on modern deep learning toolkits.

Given the practical importance of time series modeling and forecasting, a number of commercial and open-source toolkits exist. The R-forecast package (Hyndman and Khandakar, 2008) and other packages such as (Taylor and Letham, 2017; Scott and Varian, 2014) provide a plethora of models and tooling for classical forecasting methods. To the best of our knowledge, GluonTS is the first toolkit for time series modeling based on a

modern deep learning framework. Some forecasting packages in R contain neural forecasting models[8], however these pre-date modern deep learning methods and only contain stand-alone implementations of simple local models, in particular, they lack state-of-the-art architectures. As deep learning based models are gaining popularity in forecasting (Flunkert et al., to appear; Gasthaus et al., 2019; Wen et al., 2017; Laptev et al., 2017), GluonTS offers abstractions to implement many of the currently proposed models. GluonTS contains pure deep learning based models, and components from classical time series models such as Kalman filters (Seeger et al., 2017). A recent trend in the machine learning literature are fusions of deep and probabilistic models (Makridakis et al., 2018; Rangapuram et al., 2018; Smyl et al., 2018). By having probabilistic building blocks available, GluonTS allows for the systematic exploration of these models.

Most forecasting libraries that we are aware of are primarily geared towards running pre-built models and less so towards the scientific use case of model development. GluonTS differs from this by providing all the components and tools necessary for rapid model prototyping, development and benchmarking against pre-assembled models. It tries to strike a balance between a short path to production and rapid scientific exploration. The experimentation process is not as heavy-weight as in typical production forecasting systems (Böse et al., 2017), because the library has native support for local, stand-alone execution.

7. Conclusion

We introduced GluonTS, a toolkit for building time series models based on deep learning and probabilistic modeling techniques. By offering tooling and abstractions such as probabilistic models, basic neural building blocks, human-readable model logging for increased reproducability and unified I/O & evaluation, GluonTS allows scientists to rapidly develop new time series models for common tasks such as forecasting or anomaly detection. GluonTS is actively used at Amazon in a variety of internal and external use-cases (including production) where it has helped scientists to address time series modelling challenges.

GluonTS's pre-bundled implementations of state-of-the-art models allow easy benchmarking of new algorithms. We demonstrated this in a large scale experiment of running the pre-bundled models on different datasets and comparing their accuracy with classical approaches. Such experiments are a first step towards a more thorough understanding of neural architectures for time series modelling. More and more fine-grained experiments such as ablation studies and experiments with controlled data are needed as next steps. GluonTS provides the tooling necessary for such future work.

References

Martín Abadi et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1.

Felix Biessmann, David Salinas, Sebastian Schelter, Philipp Schmidt, and Dustin Lange. "Deep" learning for missing value imputation in tables with non-numerical data. In

8. For example https://cran.r-project.org/web/packages/nnfor/nnfor.pdf

Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM '18, pages 2017–2025, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6014-2.

Eli Bingham et al. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2018.

Joos-Hendrik Böse et al. Probabilistic demand forecasting at scale. *PVLDB*, 10(12): 1694–1705, 2017.

G. E. P. Box and D. R. Cox. An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, 26(2):211–252, 1964.

Nicolas Chapados. Effective Bayesian modeling of groups of related count time series. In *Proceedings of The 31st International Conference on Machine Learning*, pages 1395–1403, 2014.

Tianqi Chen et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Zhenwen Dai, Eric Meissner, and Neil D. Lawrence. MXFusion: A modular deep probabilistic programming library. In *NIPS Workshop MLOSS (Machine Learning Open Source Software)*, 2018.

Dua Dheeru and Efi Karra Taniskidou. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

Joshua V. Dillon et al. Tensorflow distributions. *CoRR*, abs/1711.10604, 2017.

James Durbin and Siem Jan Koopman. *Time series analysis by state space methods*, volume 38. OUP Oxford, 2012.

Christos Faloutsos, Jan Gasthaus, Tim Januschowski, and Yuyang Wang. Forecasting big time series: old and new. *Proceedings of the VLDB Endowment*, 11(12):2102–2105, 2018.

Valentin Flunkert, David Salinas, Jan Gasthaus, and Tim Januschowski. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, to appear.

Marco Fraccaro, Søren Kaae Sønderby, Ulrich Paquet, and Ole Winther. Sequential neural models with stochastic layers. In *Advances in neural information processing systems*, pages 2199–2207, 2016.

Jan Gasthaus. Non-parametric time series forecaster. Technical report, Amazon, 2016.

Jan Gasthaus, Konstantinos Benidis, Yuyang Wang, Syama Sundar Rangapuram, David Salinas, Valentin Flunkert, and Tim Januschowski. Probabilistic forecasting with Spline Quantile Function RNNs. In Kamalika Chaudhuri and Masashi Sugiyama, editors, *Proceedings of Machine Learning Research*, volume 89 of *Proceedings of Machine Learning Research*, pages 1901–1910. PMLR, 16–18 Apr 2019. URL http://proceedings.mlr.press/v89/gasthaus19a.html.

Agathe Girard, Carl Edward Rasmussen, Joaquin Quinonero Candela, and Roderick Murray-Smith. Gaussian process priors with uncertain inputs application to multiple-step ahead time series forecasting. In *Advances in neural information processing systems*, pages 545–552, 2003.

Felix Hieber et al. The Sockeye neural machine translation toolkit at AMTA 2018. In *AMTA (1)*, pages 200–207. Association for Machine Translation in the Americas, 2018.

J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

R. Hyndman, A. B. Koehler, J. K. Ord, and R. D. Snyder. *Forecasting with Exponential Smoothing: The State Space Approach*. Springer Series in Statistics. Springer, 2008. ISBN 9783540719182.

Rob J Hyndman and George Athanasopoulos. Forecasting: Principles and practice. *www. otexts. org/fpp.*, 987507109, 2017.

Rob J Hyndman and Yeasmin Khandakar. Automatic time series forecasting: the forecast package for R. *Journal of Statistical Software*, 2008.

Roger Koenker. *Quantile Regression*. Econometric Society Monographs. Cambridge University Press, 2005.

Rahul G Krishnan, Uri Shalit, and David Sontag. Structured inference networks for nonlinear state space models. In *AAAI*, pages 2101–2109, 2017.

Nikolay Laptev, Jason Yosinsk, Li Li Erran, and Slawek Smyl. Time-series extreme event forecasting with neural networks at Uber. In *ICML Time Series Workshop*. 2017.

Danielle C Maddix, Yuyang Wang, and Alex Smola. Deep factors with Gaussian processes for forecasting. *arXiv preprint arXiv:1812.00098*, 2018.

Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. The M4 competition: Results, findings, conclusion and way forward. *International Journal of Forecasting*, 34(4): 802 – 808, 2018. ISSN 0169-2070. doi: https://doi.org/10.1016/j.ijforecast.2018.06.001.

Xiangrui Meng et al. Mllib: Machine learning in apache spark. *CoRR*, abs/1505.06807, 2015. URL http://arxiv.org/abs/1505.06807.

Andrew Y. Ng and Michael I. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, pages 841–848. MIT Press, 2002.

Adam Paszke et al. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

Fabian Pedregosa et al. Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.*, 12: 2825–2830, nov 2011. ISSN 1532-4435.

Syama Sundar Rangapuram, Matthias Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang, and Tim Januschowski. Deep state space models for time series forecasting. In *Advances in Neural Information Processing Systems*, 2018.

Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 1530–1538. JMLR.org, 2015.

Steven L. Scott and Hal R. Varian. Predicting the present with Bayesian structural time series. *IJMNO*, 5:4–23, 2014.

Matthias Seeger, Asmus Hetzel, Zhenwen Dai, Eric Meissner, and Neil D. Lawrence. Auto-differentiating linear algebra, 2017.

Matthias W Seeger, David Salinas, and Valentin Flunkert. Bayesian intermittent demand forecasting for large inventories. In *Advances in Neural Information Processing Systems*, pages 4646–4654, 2016.

Dominique T Shipmon, Jason M Gurevitch, Paolo M Piselli, and Steve Edwards. Time Series Anomaly Detection. *arXiv:1708.03665 [stat.ML]*, page 9, 2017.

Slawek Smyl, Jai Ranganathan, and Andrea Pasqua. M4 forecasting competition: Introducing a new hybrid ES-RNN model. https://eng.uber.com/m4-forecasting-competition/, 2018.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

Sean J Taylor and Benjamin Letham. Forecasting at scale. *PeerJ Preprints*, 5:e3190v2, September 2017. ISSN 2167-9843. doi: 10.7287/peerj.preprints.3190v2.

Aäron van den Oord et al. WaveNet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016.

Ashish Vaswani et al. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.

Yuyang Wang, Alex Smola, Danielle Maddix, Jan Gasthaus, Dean Foster, and Tim Januschowski. Deep factors for forecasting. In *International Conference on Machine Learning*, pages 6607–6617, 2019.

Ruofeng Wen Wen, Kari Torkkola, and Balakrishnan Narayanaswamy. A multi-horizon quantile recurrent forecaster. In *NIPS Time Series Workshop*. 2017.

Qifa Xu, Xi Liu, Cuixia Jiang, and Keming Yu. Quantile autoregression neural network model with applications to evaluating value at risk. *Applied Soft Computing*, 49:1–12, 2016.

Guoqiang Zhang, B Eddy Patuwo, and Michael Y Hu. Forecasting with artificial neural networks:: The state of the art. *International journal of forecasting*, 14(1):35–62, 1998.

Appendix A. Experiment details and running times

Here, we provide more details about the experiments in Sec. 5 Table 1. All neural network models were trained with batch size 32, using 5000 overall batches (gradient updates). We used the ADAM optimizer with an initial learning rate of 10^{-3}. The learning rate was halved after 300 batches if there was no reduction in training loss. The gradient norm was clipped at a magnitude of 10. For DeepAR and Transformer a student's t distribution was used for the one-step ahead prediction, which works well with noisy data. For models that generate sample paths (all except CNN-QR), 100 sample paths were drawn for the evaluation. For CNN-QR the quantiles $0.1, 0.2, \ldots, 0.9$ were estimated. These quantiles were also used during the evaluation of CRPS for all methods.

The below table gives more detailed information about the datasets and the evaluation scenario used. In the table freq is the granularity of the dataset, prediction length is the forecast horizon that was used for the evaluation. Rolling evaluation indicates whether the evaluation was done over multiple forecasts. For "–" the evaluation was done for a single forecast on the last time window in each time series. In the case of rolling evaluations, a number of consecutive forecasts were evaluated shifted, by the prediction length. For instance, for electricity the last 7 days were used for evaluation where a forecast was generated for each day.

dataset	freq	prediction length	rolling window evaluation
electricity	hourly	24	7
m4-Daily	daily	14	-
m4-Hourly	hourly	48	-
m4-Monthly	monthly	18	-
m4-Quarterly	quarterly	8	-
m4-Weekly	weekly	13	-
m4-Yearly	yearly	6	-
parts	monthly	8	-
SP500-returns	business day	30	5
traffic	hourly	24	7
wiki10k	daily	60	-

Figure 3 shows the running times for the experiments in Sec. 5 Table 1.

Appendix B. Reproducibility of experiments in GluonTS

To support scientists in their work, GluonTS supports tracking the entire configuration of an experiment. This is achieved through a Python decorator `@validated` that tracks all constructor arguments of models and components. The below listing C demonstrates how to create a new neural network based model in GluonTS (see Sec. 2.6 for a discussion of the necessary components). The new estimator (`MyEstimator`) uses the `@validated` decorator and Python3 type annotations for the constructor arguments. GluonTS can serialize any class that is annotated in this way to json or to a human readable format. The `MyEstimator` has a `trainer` argument with a default value. This `Trainer` class contains parameters potentially with default values itself.

For example, if we instantiate this new estimator in an experiment

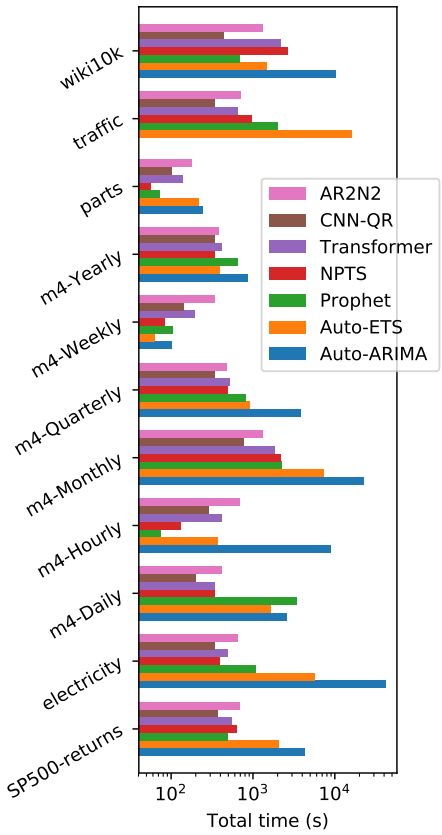

Figure 3: Running time in second for all datasets and methods (logarithmic scale). DeepAR is labeled as AR2N2 in this figure.

```
estim = MyEstimator(
  freq='D', context_length=50, prediction_length=20
)
```

and then log the instance, the following text output is generated:

```
MyEstimator(
  freq='D',
  context_length=50,
  prediction_length=20,
  act_type='relu',
  cells=[40, 40, 40],
  trainer=swist.trainer.Trainer(
    batch_size=32,
    clip_gradient=10.0,
    epochs=50,
    learning_rate=0.001,
```

```
    learning_rate_decay_factor=0.5,
    minimum_learning_rate=5e-05,
    early_stopping_patience=3,
    weight_decay=1e-08
  )
)
```

All arguments including default values and the configuration of all nested components are logged. The resulting string is not only readable for the user, it also allows the user to copy paste the code to instantiate the exact same model configuration. During a backtest GluonTS logs the entire configuration of the train/test split, the model and e.g. the evaluation configuration in this way, allowing the user to inspect all parameter settings in hindsight and to fully re-create the experimental setup.

Appendix C. Sample model code

Here, we provide a full example of a new time series model using the GluonTS abstractions, extending Sec. 2.6. Such a model can be integrated into the workflow at the beginning of Sec. 2 by replacing `DeepAREstimator` in line 11 in Listing 1 with `MyEstimator`.

```
1  from typing import List
2  from functools import partial
3  from mxnet import gluon
4  from swist.model.estimator import GluonForecastEstimator
5  from swist.forecast import QuantileForecast
6  from swist.trainer import Trainer
7  from swist import transform
8  from swist.core.component import validated
9
10 class MyTrainNetwork(gluon.HybridBlock):
11   def __init__(self, prediction_length, cells, act_type, **kwargs):
12     super().__init__(**kwargs)
13     self.prediction_length = prediction_length
14
15     with self.name_scope():
16       # Set up a network that predicts the target
17       self.nn = gluon.nn.HybridSequential()
18       for c in cells:
19         self.nn.add(gluon.nn.Dense(
20           units=c, activation=act_type
21         ))
22       self.nn.add(gluon.nn.Dense(
23         units=self.prediction_length,
24         activation=act_type
25       ))
26
```

```
27    def hybrid_forward(self, F,
28                       past_target, future_target):
29      prediction = self.nn(past_target)
30      # calculate L1 loss to learn the median
31      return (prediction - future_target).abs() \
32        .mean(axis=-1)
33
34  class MyPredNetwork(MyTrainNetwork):
35    # The prediction network only receives
36    # past_target and returns predictions
37    def hybrid_forward(self, F, past_target):
38      prediction = self.nn(past_target)
39      return prediction.expand_dims(axis=1)
40
41  class MyEstimator(GluonForecastEstimator):
42    @validated()
43    def __init__(
44      self,
45      freq: str,
46      context_length: int,
47      prediction_length: int,
48      act_type: str = 'relu',
49      cells: List[int] = [40, 40, 40],
50      trainer: Trainer = Trainer(epochs=10)
51    ) -> None:
52      super().__init__(trainer=trainer)
53      self.context_length = context_length
54      self.prediction_length = prediction_length
55      self.freq = freq
56
57      self.train_network_cls = MyTrainNetwork
58      self.pred_network_cls = MyPredNetwork
59      self.forecast_cls = partial(
60        QuantileForecast, quantiles=[0.5]
61      )
62      self.network_args = dict(
63        cells=cells,
64        act_type=act_type,
65        prediction_length=self.prediction_length
66      )
67
68    def create_transformation(self):
69      # Model specific input transform
70      # Here we use a transformation that randomly
71      # selects training samples from all series.
```

```
72      return transform.InstanceSplitter(
73        past_length=self.context_length,
74        future_length=self.prediction_length,
75      )
```