Università degli Studi di Padova

Department of Information Engineering

Master Thesis in Computer Engineering

Online Contextual System Tuning with Bayesian Optimization and Workload Forecasting

Master Candidate: Luca Moroldo

Supervisor: Prof. Nicola Ferro

XX February 2022

Academic Year 2018/2019

Summary

TODO

Index

1 Introduction

Add context and motivation: we are optimizing IT systems and we want to do that online, using the real workload, which requires bla bla presented in section state of the art.

2 Context and State of the Art

Tuning an IT system without changing its code requires to find good configurations in an enormous search space where the tunable parameters affect different layers of the IT stack, such as Operating System (OS), Database Management System (DBMS), and Java Virtual Machine (JVM). These parameters can have counter-intuitive effects with inter-dependencies, making manual optimization hard or even unfeasible when looking for the optimal configuration. Recent years have seen automatic approaches emerge [1, 2, 3, 4], with [1] using Contextual Bayesian Optimization (BO). However, when automatically tuning IT systems on a staging environment, i.e. a replica of the real system, it is still necessary to replicate the real workload to find the optimal configuration that fits the way the IT system is used at different times of the day or week. Finding such workload(s) takes time, lengthening the time required to complete the tuning process.

A possible solution to this issue is to directly tune the production system under the real workload, removing the workload analysis and replication requirement, with the performance of each configuration measured directly from the production system rather than a staging replica, which could lead to unrealistic outcomes. However, performing experiments on an IT system that is being used can be dangerous and poses new challenges: a bad configuration can cause bad user experience and consequently business issues.

This work extends the Contextual Bayesian Optimization tuner [1] to be applied in an online manner on IT systems receiving the production workload by providing two key components: a workload characterization and a workload forecasting module.

The following section briefly presents BO and section 2.1.1 presents the Contextual BO tuner of [1] explaining why workload characterization forecasting are required. The state of the art of forecasting is presented (section 2.4), followed by a section on workload forecasting in the context of IT systems (section 2.5). A general overview of workload characterization is presented in section 2.2. Clustering, that can be used for workload characterization, is described in section 2.3.

2.1 Bayesian Optimization

Bayesian Optimization [5] is a tool for the joint optimization of design choices of complex systems, such as the parameters of a recommendation system, a neural network, or a medical analysis tool. For example a typical software system made of a database, a back-end, and a front-end is characterized by an enormous amount of parameters that are often dependent on each other. Optimizing such parameters is not a simple task, and BO provides an automated approach to make such design choices.

Mathematically, the goal is to maximize (or minimize) an unknown objective function f:

$$\boldsymbol{x}^\star = \underset{\boldsymbol{x}\in\mathcal{X}}{\mathrm{argmax}} f(\boldsymbol{x}) \tag{1}$$

where $\mathcal{X}$ is the design space of interest, e.g. a compact subset of $\mathbb{R}^d$. In general, f can be any black-box function with no simple closed form that can be evaluated at any arbitrary point in the domain $\mathcal{X}$, where the evaluation can produce noise-corrupted outputs $y \in \mathbb{R}$.
BO is a sequential approach to solve equation 1: at every iteration i, the algorithm selects a new $\boldsymbol{x}_{i+1}$ at which f is queried, resulting in a measurement y_i. When the maximum number of iterations is reached, or when $y^\star$ is a satisfactory outcome, the algorithm stops returning the best configuration $\boldsymbol{x}^\star$ associated with the best outcome $y^\star$. BO is very data efficient, making it useful when the evaluations of f are costly: the model is initialized with a prior belief, and then at each iteration it is refined using observed data via Bayesian posterior updating. The acquisition function $\alpha_n : \mathcal{X} \to \mathbb{R}$

guides exploration by evaluating candidate points in $\mathcal{X}$, meaning that $\boldsymbol{x}_{i+1}$ is selected by maximizing α_n using data up to iteration i. Figure 2.1 shows a few iterations of BO. The acquisition function α_n provides a trade off between exploration and exploitation: when boosting exploration the value of α_n will be higher in areas of uncertainty, while when boosting exploitation α_n will favor locations where the surrogate model predicts a high objective. It is critical for the acquisition function to be cheap to evaluate or approximate, especially in relation to the objective function f.

In summary, BO has two key ingredients [5]: a probabilistic surrogate model, consisting of a prior distribution that captures our beliefs about f and an observation model that describes the data generation process, and a loss function that describes how optimal a sequence of queries are. The expected loss is minimized to drive the selection of $\boldsymbol{x}_i$. After observing the outcome y_i of $\boldsymbol{x}_i$, the prior is updated to produce a more informative posterior distribution.

Finally, when dealing with a family of correlated objective functions $\mathcal{T} = \{f_1, ..., f_m\}$, such as the performance of an IT system under different workloads or the same IT system running with different software versions, it may be useful to use data obtained optimizing $\{f_1, ..., f_{j-1}, f_{j+1}, ..., f_m\}$ to optimize f_j. BO has been extended to deal with such multitask scenario [6] by sharing information between the black-box functions in $\mathcal{T}$: figure 2.1 shows how data from two functions influence the posterior predictive distribution of another function.

2.1.1 Contextual Bayesian Optimization of IT systems

When applying BO to IT systems, the goal is to find a configuration $\boldsymbol{x}$ to optimize a performance indicator $y \in \mathbb{R}$ such as throughput, response time, and memory consumption [1]. If BO is applied while the system is running, it is very likely that the workload will change over time: the number of users connected to the system can increase and decrease, as their behavior can change from read-intensive to write-intensive operations. Such changes will inevitably affect how the system behaves under a specific configuration,

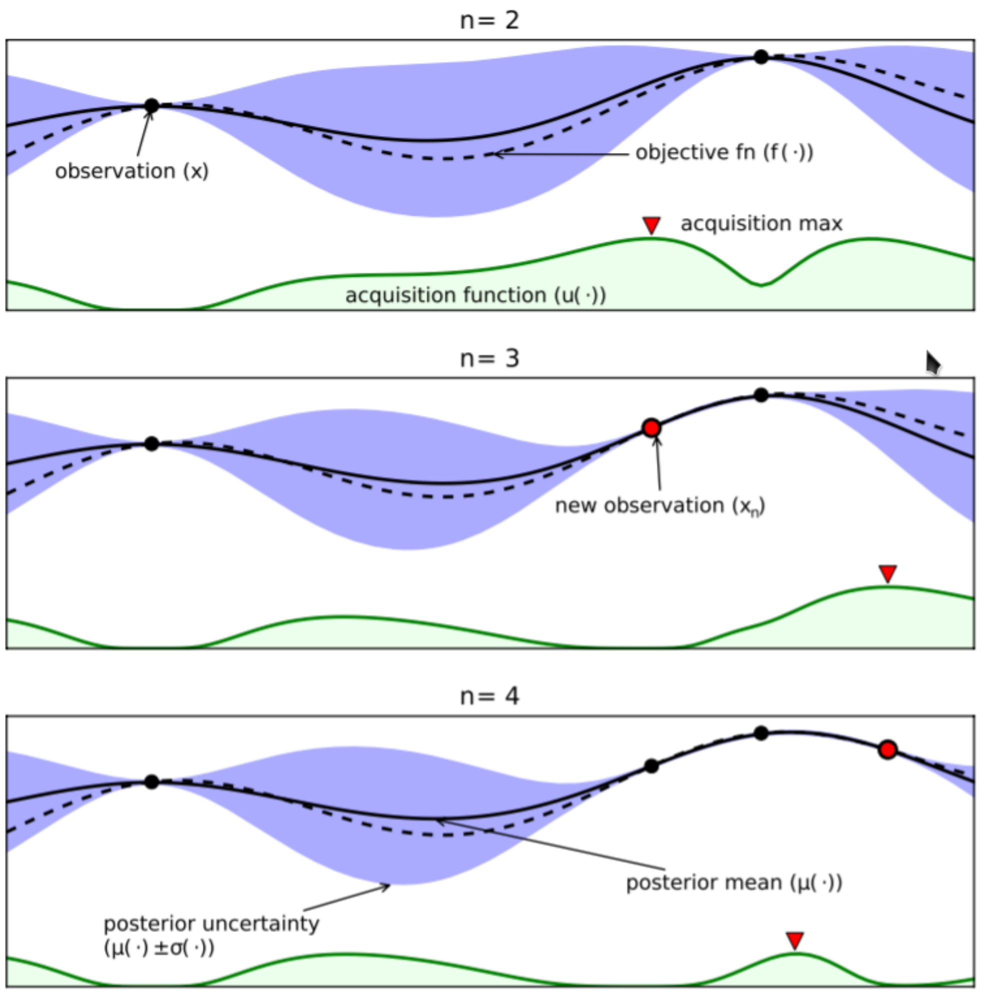


Figure 1: A few iterations of BO. The acquisition function in green guides the selection of the next point, obtaining a new observation $(\boldsymbol{x}_i, y_i)$ that updates the underlying model.

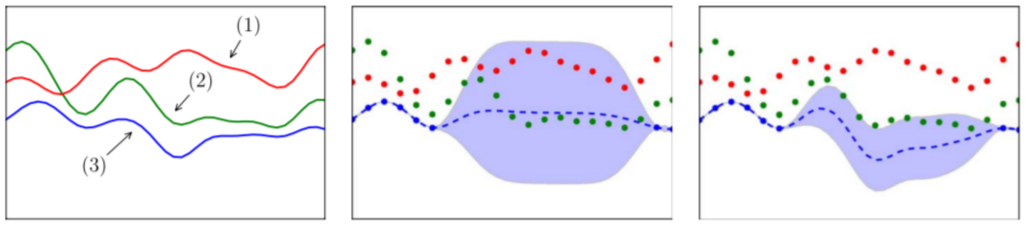


Figure 2: Multitask Bayesian Optimization. The figure in the middle shows the posterior predictive distribution of the blue function (3) without exploiting the information of the other functions.

but regardless the workload, the underlying system maintains some of its properties. This scenario perfectly fits the contextual extension of BO [6].

More formally, we are trying to maximize a function $f_{\boldsymbol{w}}$ subject to a workload $\boldsymbol{w}_t$ that changes over time. The tuning process, shown in figure 2.1.1, starts with a configuration $\boldsymbol{x}_0$ (usually the default configuration, called baseline) applied under a workload $\boldsymbol{w}_0$ whose performance indicator y_0 is measured. The tuner uses a knowledge base, initially containing only the triplet $\{(\boldsymbol{x}_0, \boldsymbol{w}_0, y_0)\}$, along with the current workload w_1 to suggest a new configuration x_1 that is applied, evaluated, and added to the knowledge base. After N iterations the tuner can exploit all information $(\{(\boldsymbol{x}_0, \boldsymbol{w}_0, y_0)\}, ..., \{(\boldsymbol{x}_N, \boldsymbol{w}_N, y_N)\})$ gathered so far to make refined suggestions.

Furthermore, when optimizing a system, it may be useful to define some constraints the system should not violate: a tuning process may be executed with the requirement satisfying some Quality of Service (QoS) levels. For example, while trying to minimize the average memory usage to reduce the infrastructure costs, the system could be expected to keep the users' requests latency below some target.
The work proposed by [1] has been extended so that it is possible to define such constraints: when a configuration violates any constraint, the violation is added to the knowledge base along along with its configuration and associated system performance. Interestingly, by penalizing violations, the tuner can be used on a system that doesn't initially satisfy some QoS levels, so that

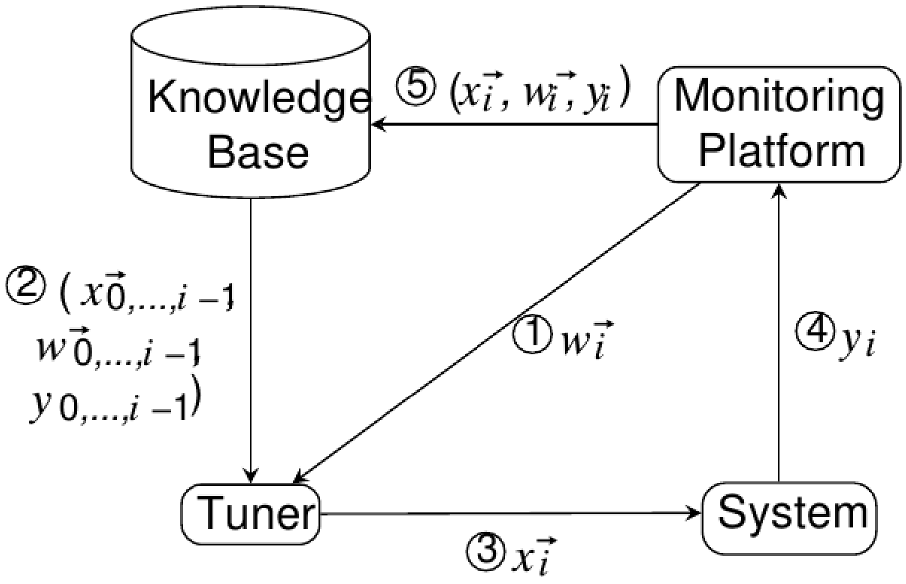

Knowledge Base
⑤ $(\vec{x_i}, \vec{w_i}, y_i)$
Monitoring Platform
② $(\vec{x}_{0,\dots,i-1}$ $\vec{w}_{0,\dots,i-1}$ $y_{0,\dots,i-1})$
① $\vec{w_i}$
④ y_i
Tuner
System
③ $\vec{x_i}$

Figure 3: BO-based tuning process. Given the workload $\boldsymbol{w}_i$, the tuner uses the knowledge base to apply a new configuration $\boldsymbol{x}_i$ that is evaluated (obtaining y_i), growing the knowledge base. TODO migliorare: disallineata

at the end of the tuning process violations are not likely to occur anymore.

The BO regression model of [1] is a Gaussian Process (GP) which derives its posterior model (i.e. the predictions) by combining observed values with its prior distribution. To make GP able to explore uncertain regions, i.e. configurations that are different from the ones in the knowledge base, it should not resort to its prior distribution [1] so that unknown configurations are not predicted to downgrade the performance of the IT system. To do so, the observed data is standardized and GP will predict that by picking a random configuration the system will exhibit a performance value that is equal to the average performance of the observed values. Nonetheless, when dealing with different workloads, it is crucial to standardize each point by taking into account the relevant workload [1]. This can be achieved by using a modified version of the Normalized Performance Improvement:

$$NPI(\boldsymbol{x}, \boldsymbol{w}) = \frac{f(\boldsymbol{x}_0, \boldsymbol{w}) - f(\boldsymbol{x}, \boldsymbol{w})}{f(\boldsymbol{x}_0, \boldsymbol{w}) - f(\boldsymbol{x}^+_{\boldsymbol{w}}, \boldsymbol{w})}$$

where $\boldsymbol{x}$ is the configuration being evaluated, $\boldsymbol{x}_0$ is the baseline configuration, $\boldsymbol{w}$ is the workload, and $f(\boldsymbol{x}^+_{\boldsymbol{w}}, \boldsymbol{w})$ is the best configuration found so far while tuning the system with the workload $\boldsymbol{w}$. Hence, [1] requires a workload characterization module (discussed in section 2.2) to cluster the workloads and effectively apply BO.

To evaluate the performance of an IT system under a new configuration, many technologies require some warm-up time. For example, the Java Virtual Machine is characterized by a lazy class loading and Just In Time (JIT) compilation that make the performance evolve over time after the Java application is launched. Usually, a window of duration that ranges from ten to thirty minutes is required for the performance to stabilize. Furthermore, after the warm-up is completed, the system performance y_i resulting from configuration $\boldsymbol{x}_i$ under workload $\boldsymbol{w}_i$ should be obtained by taking multiple measurements in order to balance the noisy environment. This evaluation process requires the workload to remain stable in order to avoid corrupting the measurements or nullifying the warm-up (e.g. a new workload may use different Java classes and functions). Therefore, a single tuning step (or ex-

periment) requires a time window $\omega_{t_1:t_2}$, starting at time t_1 and ending at time t_2, of duration $t_2 - t_1$ during which the workload w_t must be stable.
Furthermore, as mentioned before, Contextual Bayesian Optimization suggests a configuration tailored for the given workload. This means that a workload change may cause the system being optimized to under perform, potentially leading to bad QoS or even failures. Such consequences should be avoided as much as possible.

In order to predict if the upcoming tuning window will be stable the tuner requires a component capable of predicting the upcoming workload, as long as some sort of classifier that given the predicted workload indicates whether the future workload will be stable or not. Finally, once the performance of the best configuration for the given workload is satisfactory, that configuration can be applied in advance.
Forecasting and workload forecasting will be discusses in section 2.4 and 2.5 respectively.

2.2 Workload characterization

The term workload refers to all the inputs received by a given technological infrastructure [7]. Within the IT domain, understanding the properties and behavior of such workload is essential for evaluating the Quality of Service (QoS) perceived by the users in order to meet the Service Level Agreement (SLA) obligations. In such context, workload characterization provides the basis for devising efficient resource provisioning, power management, and performance engineering strategies.
By characterizing the workload and deriving workload models it is possible to summarize and explain the main properties of the workloads, generate synthetic workloads for performance evaluation studies, and define benchmark experiments. Workload characterization can be applied to different domains such as online social networks, video services, mobile devices, and cloud computing.

Characterizing the workloads requires to collect representative measurements while the system under study is operating (i.e. under the true workloads). These measurements refer to specific components of the system and capture their static and dynamic properties, along with the behavior of the users. When choosing which metrics to consider, it is important to take into account the hierarchical nature of typical infrastructures: a network sniffer provides measurements about the network traffic, logging facilities provide application-specific measurements such as the number of requests to an URL of a web application, and tracelogs contain measurements related to the resources used by jobs and tasks (e.g. CPU and memory usage). The choice of the attributes to consider for the characterization depends on its objectives and on the nature of the workload to be analyzed.

Once measurements are collected, they have to be analyzed in order to build workload models. The first step is to perform a statistical analysis to describe the properties (e.g. mean, variance, percentiles) of each attribute of interest and find any relation between them (e.g. using Pearson's correlation coefficient [8]).
A common challenge faced during this step is how to deal with outliers, that are atypical behaviors of one or more attributes: outliers could indicate previously unknown phenomena that are worth exploring, or they could correspond to anomalous operating conditions that should be discarded.

Further steps of the workload characterization methodology are multivariate analysis to analyze the components in the multidimensional space of their attributes, numerical fitting to study the dynamics of the workloads and model their temporal patterns, stochastic processes to study time-varying properties of the workloads, and graph analysis to model the behavior of interactive users [7].

Multivariate analysis allows to derive models that capture and summarize the overall properties of the workloads. A technique that has been widely used for that purpose is clustering [9], which enables unsupervised classification when labeled data is not available. A popular clustering algorithm is k-means [9], that partitions the data into k clusters identified by k centroids where each centroid would represent one type of workload. Clustering is further presented in section 2.3.
When the number of variables being analyzed is too large, it is common

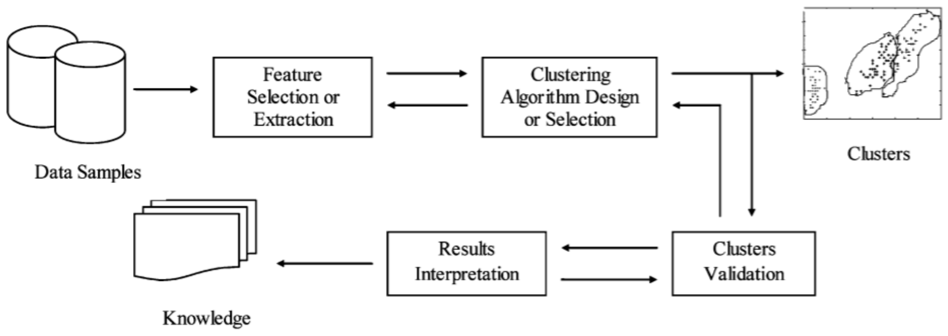

Data Samples
Feature Selection or Extraction
Clustering Algorithm Design or Selection
Clusters
Knowledge
Results Interpretation
Clusters Validation

Figure 4: Common clustering procedure.

to apply dimensionality reduction techniques such as PCA [10] to obtain a smaller set of uncorrelated variables.

With numerical fitting techniques is possible to estimate the parameters of the function that best fits the empirical data, for example to understand whether it is generated by a well-known probabilistic distribution. [11] has shown that the distribution of many workloads properties are well described by power laws (e.g. Pareto or Zipf distributions), meaning that extreme values should be investigated rather than being considered outliers.

Finally, stochastic processes such as wavelets and nonparametric filtering are used to extract trend and seasonal components from the time series representing the workload properties (see section 2.4), identify the structure of predictive models and estimate their parameters. A common goal when applying such techniques is to cope with capacity planning and resource management.

2.3 Clustering

In the era of data, clustering provides a way to classify and group information into a set of categories, called clusters, when labels are not provided. The goal is to learn a new object or understand a phenomenon, and try to seek the features that can describe it in order to make comparisons with other known objects or phenomena [9]. Such learning process is called unsupervised learning. In general, a cluster is described by considering its internal homogeneity and the external separation [12], meaning that the same cluster should present similar patterns, while patterns in different clusters should not.

Formally, given an input set $X = \{\boldsymbol{x}_1, ..., \boldsymbol{x_N}\}$ with $\boldsymbol{x}_i \in \mathbb{R}^d$, *hard clustering* attempts to seek a K-partition of X, $C = \{C_1, ..., C_K\}$ with $K \leq N$ such that:

1. $C_i \neq \emptyset$ for $i = 1, ..., K$
2. $\cup_{i=1}^{K} C_i = X$
3. $C_i \cap C_j = \emptyset$ for each $i, j = 1, ..., K$ with $i \neq j$

Therefore, at the end of the procedure, each point in X belongs to a single cluster. In *fuzzy clustering* this constraint is relaxed, and a point $\boldsymbol{x}_i \in X$ can belong to multiple clusters with a certain degree of membership [13]. Another alternative to hard clustering is *hierarchical clustering*, that repeatedly agglomerates points (or, symmetrically, divides clusters) creating a dendrogram [14].

The general clustering procedure is depicted in figure 2.3. The feature selection or extraction step chooses a subset of features from the set of available features. From that subset new features can be generated, for example by using Principal Component Analysis. This step can heavily affect the clustering result. Then, one or more clustering algorithms must be selected often along with a proximity measure (e.g. the Euclidean distance). Note that there isn't any clustering algorithm that is capable of solving all types of problems [15]. Finally, each clustering algorithm must be objectively validated, and the results interpreted.

In the context of workload characterization (see section 2.2) we are interested in hard clustering methods to automatically group workloads.
A well-known clustering method is k-means, that partitions X into $k \leq |X|$ clusters $C_1, ..., C_k$ by finding:

$$\underset{C_1,...,C_k}{\text{argmax}} \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in S} ||\boldsymbol{x} - \boldsymbol{\mu}_i||^2$$

where $\boldsymbol{\mu}_i$ is the *centroid* of cluster C_i, and a point $\boldsymbol{x}$ belongs to the cluster having the closest centroid. In order to effectively initialize the value of μ_i, k-means++ can be used [16]. k-means requires the number of clusters to be given as input.

Mean shift [17] is another centroid-based clustering procedure that at each iteration moves the points towards the direction of maximum density by using a kernel function $K(x_i, x)$ (e.g. a Gaussian kernel) that controls the direction of the movement $m(x)$:

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x) x_i}{\sum_{x_i \in N(x)} K(x_i - x)}$$

where $N(x)$ is the neighborhood of x. The algorithm stops on convergence, i.e. when points cannot further be moved or the movement is below some threshold ϵ. Unlike k-means, mean shift doesn't require to know the number of clusters, but it must be provided with a bandwidth parameter that determines the size of the neighborhood $N(x)$. Such bandwidth can be estimated using nearest neighbors.

OPTICS [18] is a density-based approach that requires two parameters: the minimum number of points required to form a cluster, and the radius ϵ to consider when forming clusters. OPTICS is very data-efficient for small values of ϵ, and can be used for outlier detection as it doesn't necessarily need to assign a cluster to each point.

Finally, Gaussian Mixture Models (GMM) [19] provide a probabilistic approach to clustering by estimating k Gaussian distributions via Expectation-Maximization . The expectation step calculates the probability that a point $\boldsymbol{x}_i$ belongs to the cluster C_j, and the maximization step updates the parameters (mean and covariance matrix) of the distributions representing the clusters in order to maximize the log-likelihood function. This process is repeated until convergence. To do so, GMM require to be given the number of components (i.e. clusters) and the covariance type. When compared to k-means, the advantage of GMM is that it is not limited to spherical-shaped clusters.

When evaluating the quality of a clustering $C = \{C_1, ..., C_k\}$, the Silhouette coefficient [20] computes a score representing how well-separated the clusters are. Similarly, the Bayesian Information Criterion [21] is an alternative applicable to GMM. These scores can be used to choose the number of clusters to be found or method-specific parameter.

2.4 Forecasting

Forecasting is a common data science task that makes use of temporal data [22] to help organizations with capacity planning, goal setting, and anomaly detection. It is required in many situations: for example, deciding whether to build another warehouse in the next five years requires forecasts of future demand, and scheduling staff in a call center next week requires forecasts of call volumes.

The first successful forecasting methods have been proposed around *1950*, some of them being Exponential Smoothing [23] and ARIMA [24], which originated a wide variety of derived techniques [25]. In the big-data era, where companies have huge numbers of time series each with their own characteristics, traditional techniques have shown some limitations due to specific model requirements (section 2.4.1 and 2.4.2), model inflexibility, necessity of manual feature engineering, lack of automation, difficulties of dealing with missing data, and lack of well-performing multivariate models [25].
Recent developments have seen pure deep learning models joining the fields with inconsistent performance [26], but highlighting the possibility of exploiting huge datasets in order to learn a single global model capable of recognizing complex and sometimes shared time series patterns. Other recent deep learning research achievements, especially in the natural language processing domain [27, 28, 29], have inspired promising models [30, 31, 26], some of them having an hybrid architecture that utilizes both statistical and machine learning (ML) features [32, 33]. Interestingly, the winner of the 2018 M4 competition [32] was a combination of Exponential Smoothing and deep learning [34], while the top-performing submissions of the 2020 M5 competition [35], where most of the time series have some kind of correlation and share frequency and domain, cross-learning ML models have shown their

potential with the top-performing submissions using a weighted average of several pure ML models.
Other methods, such as the one proposed by Prophet [36], provide an analyst-in-the-loop approach suggesting that by injecting domain-specific knowledge into the model it's still possible to outperform fully automated approaches, especially with small amounts of data.
Nevertheless, Artificial Neural Network (ANN) based models have only recently started overtaking simpler classical models [32, 35] opening a set of inspiring possibilities, and the market has seen big companies developing their own solutions [36, 33, 37, 34] highlighting the necessity for the businesses of better forecasting techniques.

Despite the forecasting importance, there are still serious challenges associated with producing reliable and high quality forecasts: time series often have long term dependencies with nonlinear relationships. Moreover, the quality of forecasts are heavily affected by model selection, model tuning, and covariates (e.g. dynamical historical features) selection, where the data scientist has to manually inspect data and inject domain knowledge into the model [26, 36].
The necessity of tailored forecasting models comes from the fact that time series can be very different from each other, exhibiting complex patterns and relationships with other time series and data in general.

Nevertheless, time series can often be seen as a composition of a trend, a seasonal pattern, and a cycle [38]. A trend exists if there is a long-term increase or decrease in the data, which can be linear or not, and can be subject to changes that increase or decrease the trend. A seasonal pattern occurs when a time series is affected by seasonal factors like the hour of the day or the day of the week, with fixed and known frequency. Finally, a cycle occurs when data rises and falls without a fixed frequency, e.g. due to economic conditions. Cycles are usually longer than seasonal patterns and have more variable magnitudes.

Trend and cycles are usually combined into a single trend-cycle component, often referred as trend for simplicity. Therefore, we can think of a time series as a combination of a trend-cycle component, a seasonal component,

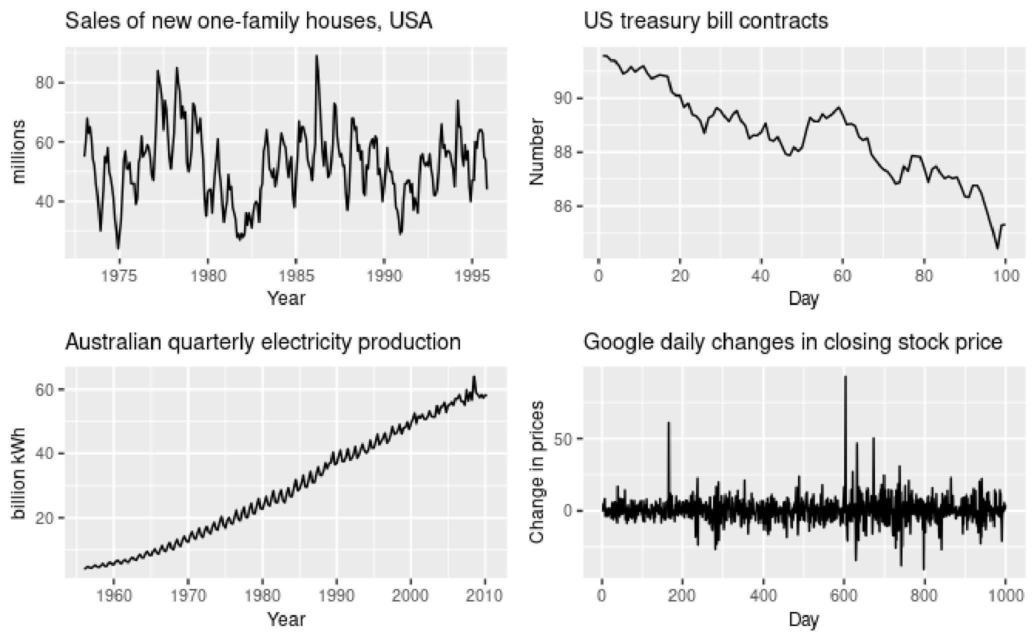


Figure 5: Four time series examples. Top left: seasonality and cycle; top right: trend only; bottom left: trend and seasonality; bottom right: random fluctuations

and a remainder component containing anything else in the time series. By assuming additive decomposition we can write:

$$y_t = S_t + T_t + R_t$$

where y_t is the time series, S_t the seasonal component, T_t the trend component, and R_t the remainder component, all at period t. When considering multiplicative decomposition, which occurs when the variation of the trend or of the seasonal component is proportional to the time series level, we can write:

$$y_t = S_t * T_t * R_t$$

To obtain such decomposition the Seasonal and Trend decomposition using Loess(STL) [39] can be applied, leading to the separation of trend, seasonal-

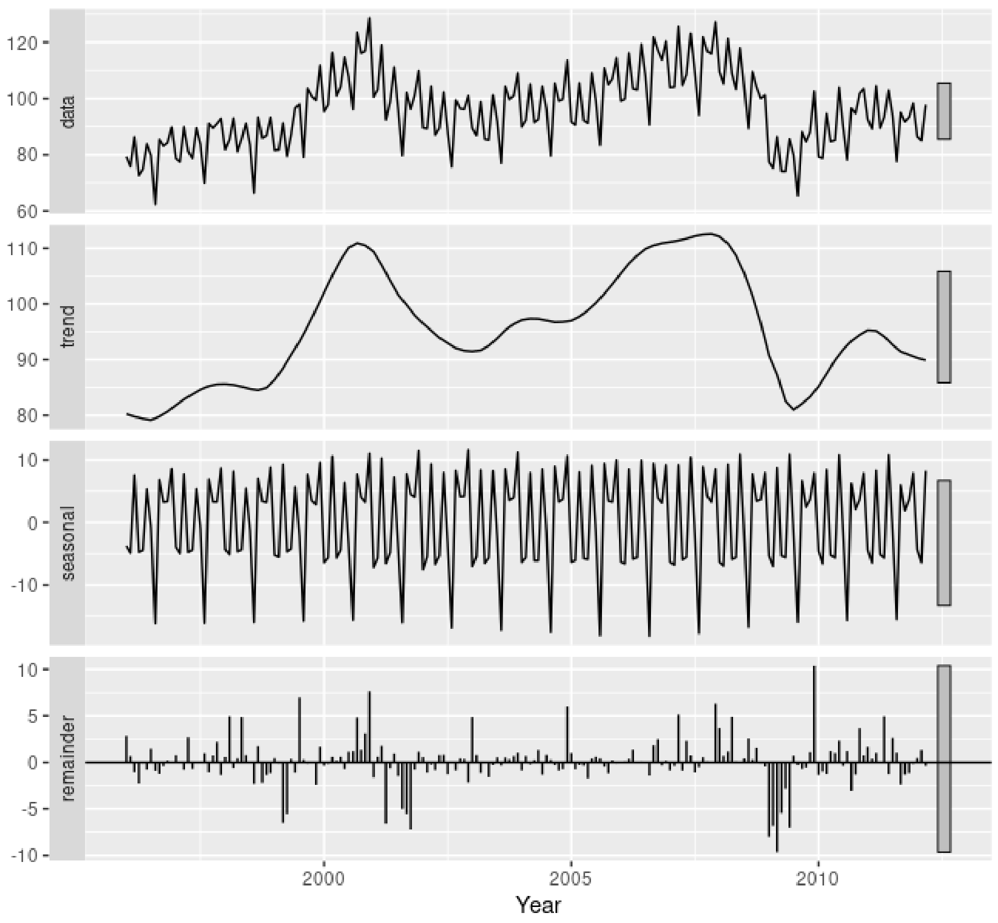


Figure 6: Additive decomposition of a time series using STL

ity, and remainder as shown in figure 6.

Given this background, let $Z = \{z_{i,1:T_i}\}_{i=1}^{N}$ be a set of N univariate time series where $z_{i,1:T_i} = (z_{i,1}, z_{i,2}, ..., z_{i,T_i})$ and $z_{i,t} \in \mathbb{R}$ is the value of the i-th time series at time t. The time series in Z may have different sampling frequencies, can start at different times, and may have missing values. Furthermore, let $X = \{\boldsymbol{x}_{i,1:T_i+\tau}\}_{i=1}^{N}$ be a set of associated, time-varying covariate vectors with $\boldsymbol{x}_{i,t} \in \mathbb{R}^D$ being any useful information that must be known before computing the forecast up to time $T_i + \tau$ (e.g. a holiday flag).
The goal of forecasting [38] 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 $\boldsymbol{x}_{i,1:T_i+\tau}$, and the

Category	Modeling
Generative	$p(z_{i,1:T_i+\tau} \vert \boldsymbol{x}_{i,1:T_i+\tau}, \Phi)$
Discriminative	$p(z_{i,T_i+1:T_i+\tau} \vert z_{i,1:T_i}, \boldsymbol{x}_{i,1:T_i+\tau}, \Phi)$

Table 1: Forecasting models

model parameters Φ:

$$p(z_{i,T_i+1:T_i+\tau}|z_{i,1:T_i}, \boldsymbol{x}_{i,1:T_i+\tau}, \Phi) \tag{2}$$

which, depending on the model, can be reduced to point forecast by considering the mean (e.g. μ if the model uses a Gaussian distribution), the median, or by drawing Monte Carlo samples to approximate the mean. The choice between probabilistic and point forecast depends on the application: probabilistic forecast can be used for anomaly detection or when the task has an asymmetric cost for over and under-predicting.
Equation 2 is a supervised learning problem where the model structure is usually fixed upfront and we want to learn the model parameters Φ using an optimization method such as maximum likelihood estimation.

Univariate models learn the model parameters Φ for each individual time series, while multivariate models are capable of learning a single global model for multiple time series by sharing the parameters.
As noted by the latest M5 competition [35], nowadays time series models are typically sufficient for identifying and capturing their historical data pattern, i.e. level, trend, and seasonality. However, relying solely on historical data fails to effectively account for the effects of holidays and special events. Moreover, such factors can affect historical data, leading to distorted time series and consequently models. In such settings, the information from exogenous/explanatory variables, i.e. the covariates $\boldsymbol{x}_{i,1:T_{i+\tau}}$, becomes of critical importance to improve accuracy; in fact, recent models such as the ones discussed later [36, 30, 31] allow the inclusion of these kind of variables.

Time series models can be categorized as generative and discriminative [40] (table 2.4): generative models assume that time series data is generated by an unknown stochastic process with some parametric structure of parameters Φ given the covariates X, while discriminative models model the conditional distribution for a fixed horizon τ. Discriminative models are typically more flexible since they make less structural assumptions [33].

2.4.1 Exponential Smoothing

Exponential smoothing [23] is one of the oldest forecasting techniques that belongs to the generative models class. It is a simple and lightweight state-space model that smooths random fluctuations by using declining weights on older data, it's easy to compute and requires minimum data. The simplest form of exponential smoothing assumes that all past observations have equal importance:

$$\hat{y}_{T+1} = \tfrac{1}{T} \textstyle\sum_{t=1}^{T} y_t$$

where $\hat{y}_{T+1}$ is the forecasted value at time $T + 1$ knowing past values up to time T. By introducing decaying weights, the formula becomes:

$$\hat{y}_{T+1} = A(y_T + By_{T-1} + B^2 y_{T-2} + B^3 y_{T-3} + ...)$$

where $A \in [0, 1]$ and $B = 1 - A$, A can attenuate the effect of old observations. The formula can be recursively applied to obtain observation at $T + k$, $k > 1$.

Exponential smoothing has been extended to take into consideration linear trend and seasonality [23]. The trend can be approximated applying the same equation above on the time series $z_t = y_t - y_{t-1}$ while to model seasonality its period must be known beforehand.

Note that due to its nature the forecasts produced by Exponential Smoothing will lag behind the actual trend.

A more complex state-space approach, called Innovation State Space Model (ISSM) [41], has been proposed to add a statistical model that describes the data generation process, therefore providing prediction intervals.

ISSM maintains a latent state vector $\boldsymbol{l}_t$ with recent information about level, trend and seasonality which evolves over time adding a small innovation at each time step (i.e. the Gaussian noise):

$$\boldsymbol{l}_t = F_t \boldsymbol{l}_{t-1} + \boldsymbol{g}_t \epsilon_t, \ \epsilon_t \sim \mathcal{N}(0, 1)$$

where $\boldsymbol{g}_t$ controls innovation strength and F_t is the transition matrix. The observations become a linear combination of the current state $\boldsymbol{l}_t$:

$$y_{t+1} = \boldsymbol{a}_t^T \boldsymbol{l}_t + b_t + \nu_t, \ \nu_t \sim \mathcal{N}(0, 1)$$

The ISSM parameters $(F_t, \boldsymbol{g}_t, \boldsymbol{a}_t, b_t, \nu_t)$ are typically learned using the maximum likelihood principle.

2.4.2 ARMA models

ARMA models are Auto-Regressive models with a Moving-Average component, they provide a complementary approach to Exponential Smoothing and they belong to the generative models class.
The Auto Regressive component of order p, AR(p), predicts the next value using a linear combination of p previous known values, while the Moving Average component of order q, MA(q), takes into consideration the average and the last q differences between the predicted and the actual value. When combined, they form an ARMA(p, q) model:

$$\hat{y}_t = \text{ARMA}(p, q) = \text{AR}(p) + \text{MA}(q) = \textstyle\sum_{i=1}^{p} \phi_i y_{t-i} + \mu + \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i}$$

Unfortunately, ARMA requires the time series to be stationary, i.e. without trends and seasonality. To make a time series stationary and apply ARMA models, Box and Jenkins [42] proposed an approach by: (1) providing guidelines for making the time series stationary, (2) suggesting the use of autocorrelations and partial autocorrelation for determining appropriate values for p and q, (3) providing a set of computer programs to identify appropriate values for p and q, and (4) estimating the parameters involved. The approach is known as the Box-Jenkins methodology to ARIMA models, where the letter ‘I ’means “Integrated”, reflecting the need for differencing the time series to make it stationary. Furthermore, ARIMA can deal with seasonality by applying seasonal differencing, but requires the seasonality period to be known beforehand. This extension is called SARIMA.

The more general SARIMA(p, d, q, P, D, Q, m) model is defined by 7 parameters:

- p: trend auto-regression order
- d: trend difference order
- q: trend moving average order
- P: seasonal auto-regressive order
- D: seasonal difference order
- Q: seasonal moving average order
- m: seasonal period steps

Besides the availability of the well defined 3-steps Box-Jenkins framework, a decent amount of human work was still required to find the right values for these parameters. To tackle this issue many approaches have been developed, some of them being made available only by commercial software. A well known automated solution has been implemented by [43], where they make use of unit root tests to find the differencing orders and the Akaike’s information criterion (AIC) to select the best combination of p, q, P, and Q. AIC introduces a model complexity penalty to avoid overfitting data.
Nevertheless, the number of steps of the seasonal period must still be given by the analyst, although it’s often a well known seasonality (e.g. daily, weekly, yearly).

The vector ARIMA (VARIMA) model has been proposed as a multivariate generalization of the univariate ARIMA model, but in general Vector Auto-Regressive (VAR) models tend to suffer from overfitting, providing poor out-of-sample forecasts [25].

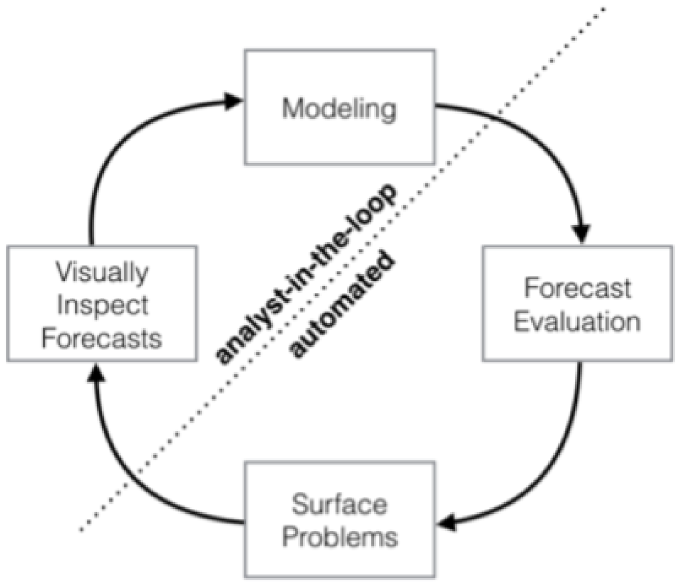


Figure 7: Prophet analyst-in-the-loop approach

2.4.3 Prophet

Prophet [36] is a solution developed by Facebook that provides an "analyst-in-the-loop" approach (figure 7) and a flexible model that fit a wide range of business time series. The model simplifies the process of adding domain knowledge about the data generation process and reduces the time required to obtain high quality forecasts. Finally, Prophet is able to automatically handle time series with trend changes, multiple seasonality, and holidays effects. Prophet model is a Generalized Additive Model (GAM) [44] that decomposes trend, seasonalities, and holidays combining them in the following equation:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t$$

where $g(t)$ is the trend function which models non-periodic changes, $s(t)$ is

the seasonality function that models periodic changes like daily and weekly seasonality, and $h(t)$ represents the effect of holidays which occur on potentially irregular schedules. Finally, the error term ϵ_t represents anything not accommodated by the model, which is assumed to be normally distributed.
The GAM formulation can be easily extended to add new components as necessary and it can fit very quickly using optimization methods like L-BFGS [45], making the forecasting problem a curve-fitting problem which allows non-regularly spaced measurements (e.g. due to missing values).
The trend model $g(t)$ can be a saturating growth model capable of dealing with a limited population growth (e.g. the number of subscriptions limited by the population of a country) or a piece-wise linear model. The latter has the following form:

$$g(t) = (k + \boldsymbol{a}(t)^T\boldsymbol{\delta})t + (m + \boldsymbol{a}(t)^T\gamma)$$

where k is the growth rate, $\boldsymbol{\delta}$ is the rate adjustments vector, m is the offset parameter, and γ_j is set to $-s_j\delta_j$ to make the function continuous.
Therefore, $\boldsymbol{\delta} \in R^S$ defines S trend change points occurring at time s_j, with δ_j being the rate adjustment. The rate at time t is $k + \sum_{j:t>s_j} \delta_j$, which is more cleanly defined by a vector $\boldsymbol{a}(t) \in \{0,1\}^S$ such that:

$$a_j(t) = \begin{cases} 1 & \text{if } t \geq s_j \\ 0 & \text{otherwise} \end{cases}$$

that makes the rate at time t be $k + \boldsymbol{a}(t)^T\boldsymbol{\delta}$
The change points s_j can be specified by the analyst or they can be automatically selected by putting a sparse prior on $\boldsymbol{\delta}$, e.g. a Laplace prior.
The seasonality function $s(t)$ is modeled using Fourier series, meaning that a seasonality with period P can be approximated by:

$$s(t) = \textstyle\sum_{n=1}^{N}(a_n \cos(\frac{2\pi nt}{P}) + b_n \sin(\frac{2\pi nt}{P}))$$

which can be automatically estimated finding $2N$ parameters, i.e.

$\boldsymbol{\beta} = (a_1, b_1, ..., a_N, b_N)$. By choosing N, the series can be truncated at different depths allowing to fit seasonal patterns that change more or less quickly, possibly leading to overfitting. Finally, the initialization $\boldsymbol{\beta} \sim \mathcal{N}(0, \sigma^2)$ allows to impose a prior on the seasonality by choosing σ.
The holiday model $h(t)$ can deal with predictable shocks that cannot be modeled by a smoothed Fourier series. An example could be the increase of units sold during Easter, which doesn't falls a specific day. An analyst can provide a list of dates of interest D_j for each holiday j, so the holiday model becomes:

$$h(t) = Z(t)\boldsymbol{\kappa}$$

with $Z(t) = [\mathbf{1}(t \in D_1), ..., \mathbf{1}(t \in D_L)]$ and $\boldsymbol{\kappa}$ initialized as $\boldsymbol{\kappa} \sim \mathcal{N}(0, v^2)$, like it was done with seasonality.
The Prophet solution is capable of achieving lower prediction errors when compared to traditional methods like Exponential Smoothing and ARIMA, with very quick fitting time.

2.4.4 ML models

The forecasting field has seen past practitioners proposing novel Neural Networks (NN) architectures that could not be considered competitive against simpler univariate statistical models. However, we are now living in the Big Data era: companies have gathered huge amounts of data over the years containing important information about their business patterns, unlocking the possibility of learning effective multivariate models. Big data in the context of time series doesn't necessarily mean having single time series with with a lot of historical data, but it rather means that there are many related time series from the same domain [46]. In such context, models capable of learning from multiple time series have emerged [35] outperforming traditional ones while alleviating the time and labor intensive manual feature engineering by making no explicit assumptions on data and therefore being more flexible when compared to traditional techniques such as ARIMA and Exponential Smoothing.

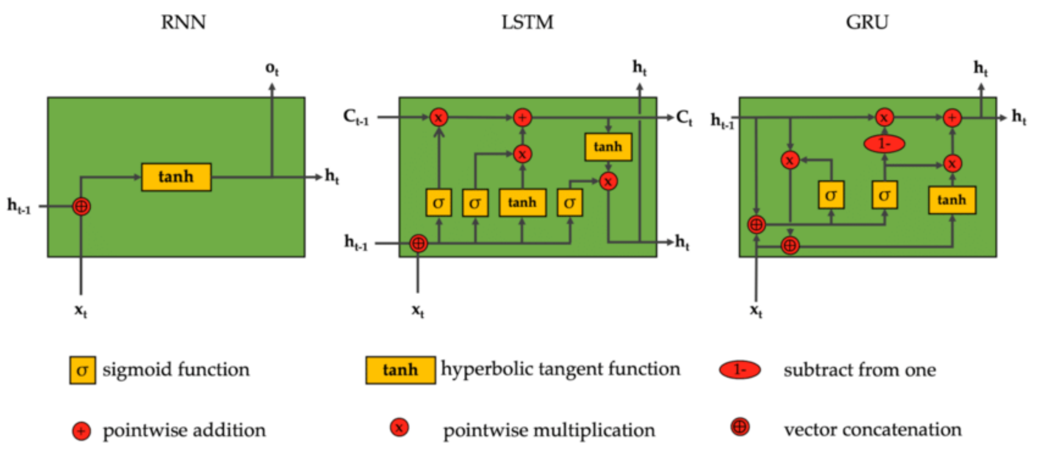


Figure 8: Architecture of an Elman recurrent unit, LSTM, and GRU. The output at step $t-1$ influences the output at step t.

All recent successful models are based on Recurrent Neural Networks (RNN) [47, 46], which demonstrated state-of-the-art performance in various applications handling sequential data like text, audio, and video where some kind of state must be kept while processing. RNNs can be combination of recurrent units like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) (see figure 8). By using recurrent edges connecting adjacent time steps (i.e. feedback loops), RNNs introduce the notion of time to the model: at time step t the network receives the current input $\boldsymbol{x}^{(t)}$ plus the previous network state $\boldsymbol{h}^{(t-1)}$ producing a new context $\boldsymbol{h}^{(t)}$ (often called hidden state) and eventually an output $\boldsymbol{o}^{(t)}$. The context acts as a memory of what the network has seen so far and influences the output, unlocking a stateful decision making.
However, the first recurrent unit (i.e. the Elman recurrent unit) suffered from the vanishing/exploding gradient problem [48] which causes the inability of carrying long-term dependencies. To address this shortcoming, the Elman recurrent unit has been extended leading to improved variants such as LSTM and GRU.

LSTM uses two components for its state: the hidden state and the internal state, containing short-term and long-term memory respectively. Furthermore, LSTM introduces a gating mechanism made of an input, forget,

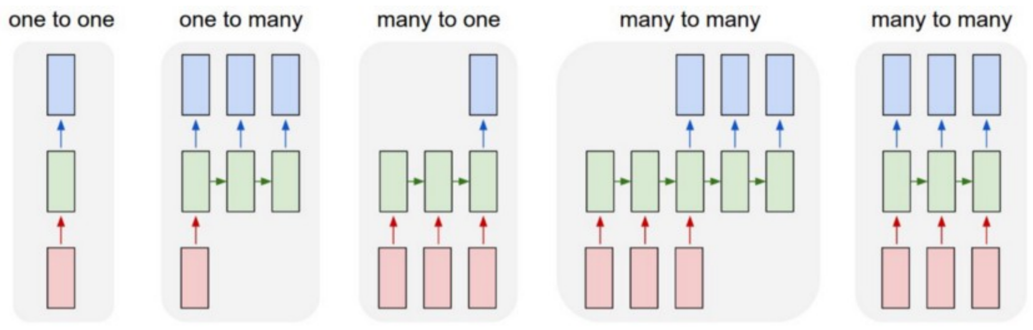


Figure 9: RNN architectures. Red rectangles are input vectors, blue rectangles are output vectors, and green rectangles are recurrent units such as LSTM or GRU which share the same weights while their state evolves from left to right.

and output gate used to filter what should or should not be kept of the state in the next step (for example, to disable the output contribution to the LSTM state just set the output gate to zero). GRU is a simpler version of LSTM with fewer gates (update and reset) which allows faster computations and is less prone to overfitting due to the lower number of parameters.

Recurrent units (e.g. LSTM, GRU) can constitute RNNs in various types of architectures, depending on the application: a many-to-one (or sequence-to-vector) architecture can be used for sentiment classification, one-to-many (or vector-to-sequence) for for music generation, and many-to-many (or sequence-to-sequence) for machine translation (see figure 9).

To obtain such architectures a single LSTM can be used, in that case the green LSTM in figure 9 is unfolded such that in the whole processing of the input the same weights are used, while the internal state of the LSTM evolves. Nevertheless, multiple LSTMs can be stacked together such as in figure 10 composing multiple layers and increasing the expressiveness of the network. Usually when forecasting the size d of the output (or the internal state of an LSTM) doesn't match the dimension of the forecasting horizon H. In such cases, another neural layer is added to map $\boldsymbol{o}^{(t)} \in \mathbb{R}^d$ to the forecast $\hat{\boldsymbol{y}}^{(t)} \in \mathbb{R}^H$. This neural layer is trained together with the LSTM, with the loss (e.g. the forecasting error $|\hat{y} - y|$) being calculated per each time step and accumulated until the end of the time series after which backpropagation

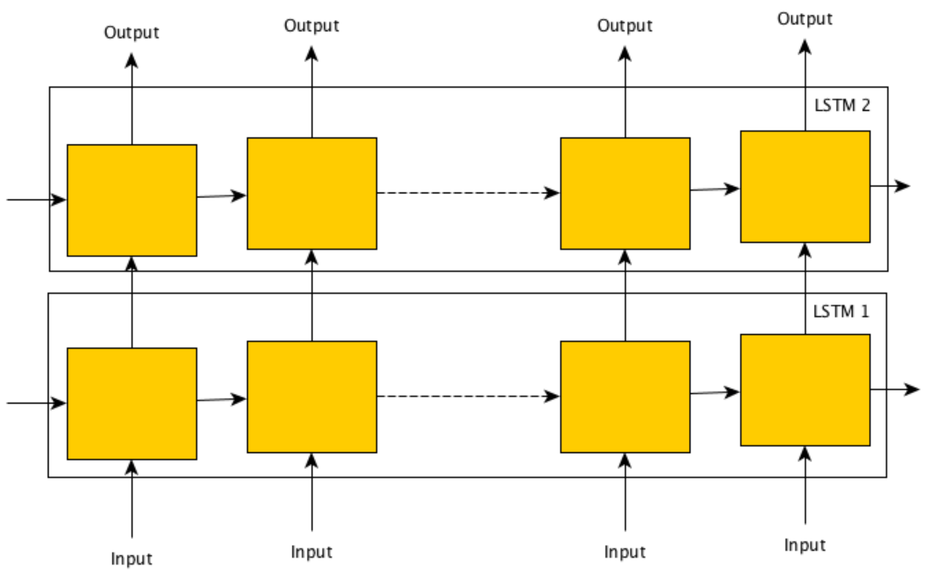


Figure 10: 2-layers stacked LSTMs.

through time is executed.

Forecasting requires a many-to-many architecture: the input is a sequence, i.e. a time series, and the output is another sequence that is a continuation of the input sequence, i.e. a forecast of horizon H. In such context, the Sequence to Sequence (S2S) [28] models have proven to be successful. S2S models are made of two RNNs: an encoder followed by a decoder, as shown in figure 11. The encoder is used to extract features from known time series data in order to produce a context vector (e.g. the LSTM hidden state) that is given as input to the decoder to produce forecasts. Examples of models based on S2S are DeepAR [30] and Multi-Horizon Quantile Recurrent Forecaster [49], explained later. By defining an encoder and a decoder, a model is allowed to see a limited amount of past values and can predict a fixed horizon, which means that any context and prediction length change requires re-training.

The decoder at time step $t+1$ can receive as input the prediction made at step t, but many forecasting problems have long periodicity (e.g. 365 days) and may suffer memory loss during forward propagation. To overcome the

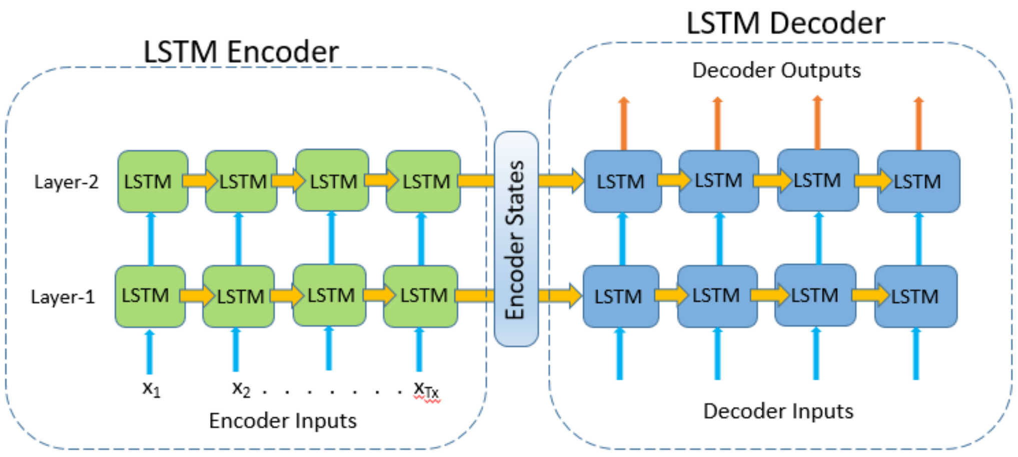


Figure 11: Sequence to sequence model

long-term dependency issue [50] proposed a recurrent unit which computes a hidden state $\boldsymbol{h}^{(t)}$ not only based on previous state $\boldsymbol{h}^{(t-1)}$ but also a specific set of other past states (e.g. $(\boldsymbol{h}^{(t-2)}), ..., \boldsymbol{h}^{(t-D)}))$ facilitating the ability of keeping long dependencies. This technique is called skip-connection. However, the naive alternative adopted by [30, 49] obtains the same effect by directly feeding past time series values $(y_{t-1}, ..., y_{t-D})$ as feature inputs to the decoder.

More generally, the idea of feeding both time-dependent and time-independent features as input (often called exogenous variables), along with the time series data points, has proven to be successful: when dealing with huge datasets, assigning time-independent (or static) features such as the category of the time series (e.g. “clothing” in the context of shopping) allow the model to learn both global and category-specific patterns, while time-dependent (or dynamic) features like day of week, holidays, and relevant events allow the model to learn and distinguish seasonality patterns from one-shot events like anomalies, reducing the risk of overfitting. However, such dynamic features must be known beforehand when computing forecasts, and in some contexts they can be used to make conditional forecasts, e.g. ”*How many units of product X will I sell if I set the price to Z?*”.

The usage of such combination of features facilitates the learning of a global model exploiting information from many time series simultaneously.

For NNs this means that weights are learned globally, but the state is maintained for each time series. Furthermore, the global model can be used to forecast time series that have never been seen during training and lack of data, as the model can still use patterns learned from the training set.

2.4.5 DeepAR

DeepAR [30] is a discriminative model capable of probabilistic forecasts in the form of Monte Carlo samples, it is based on Sequence to Sequence (S2S) [28] and can learn a global model from multiple time series.
Alongside with a model, DeepAR proposes a solution to the issue of dealing with time series having very different magnitudes, which are known to ruin the learning of an effective global model reducing the effectiveness of normalization techniques on some datasets [30].

DeepAR goal is to model the conditional distribution

$$P(\boldsymbol{z}_{i,t_0:T}|\boldsymbol{z}_{i,1:t_0-1}, \boldsymbol{x}_{i,1:T})$$

where $\boldsymbol{z}_{i,t_0:T} = [z_{i,t_0}, z_{i,t_0+1}, ..., z_{i,T}]$ is the future (or prediction range), $\boldsymbol{z}_{i,1:t_0-1}$ is the past (or conditioning range), and $\boldsymbol{x}_{i,1:T}$ are covariates that must be known for all time points.

DeepAR assumes that its distribution $Q_\Theta(\boldsymbol{z}_{i,t_0:T}|\boldsymbol{z}_{i,1:t_0-1}, \boldsymbol{x}_{i,1:T})$ consists of a product of likelihood factors:

$$Q_\Theta(\boldsymbol{z}_{i,t_0:T}|\boldsymbol{z}_{i,1:t_0-1}, \boldsymbol{x}_{i,1:T}) = \prod_{t=t_0}^{T} Q_\Theta(z_{i,t}|\boldsymbol{z}_{i,1:t-1}, \boldsymbol{x}_{i,1:T}) = \prod_{t=t_0}^{T} l(z_{i,t}|\theta(\boldsymbol{h}_{i,t}, \Theta))$$

which is parametrized by the output $\boldsymbol{h}_{i,t}$ of an autoregressive RNN

$$\boldsymbol{h}_{i,t} = h(\boldsymbol{h}_{i,t}, \boldsymbol{z}_{i,t-1}, \boldsymbol{x}_{i,t}, \Theta) \tag{3}$$

where h is implemented by multi-layer RNN with LSTM cells, meaning that $\boldsymbol{h}_{i,t}$ is given by the internal state of the LSTMs as shown in figure 12.
$l(z_{i,t}|\theta(\boldsymbol{h}_{i,t})$ is the likelihood of a fixed distribution (e.g. Student's t-distribution) whose parameters are given by a function $\theta(\boldsymbol{h}_{i,t}, \Theta)$ of the network output

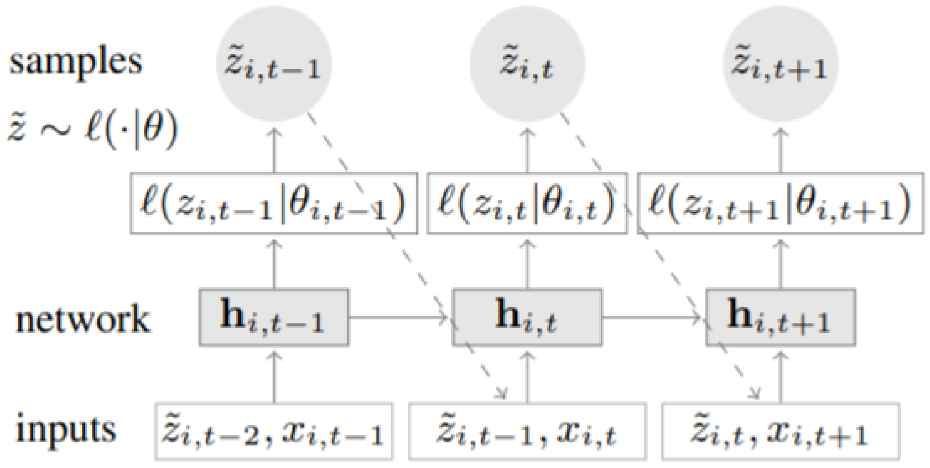

samples
$\tilde{z} \sim \ell(\cdot|\theta)$
$\tilde{z}_{i,t-1}$
$\tilde{z}_{i,t}$
$\tilde{z}_{i,t+1}$
$\ell(z_{i,t-1}|\theta_{i,t-1})$
$\ell(z_{i,t}|\theta_{i,t})$
$\ell(z_{i,t+1}|\theta_{i,t+1})$
network
$\mathbf{h}_{i,t-1}$
$\mathbf{h}_{i,t}$
$\mathbf{h}_{i,t+1}$
inputs
$\tilde{z}_{i,t-2}, x_{i,t-1}$
$\tilde{z}_{i,t-1}, x_{i,t}$
$\tilde{z}_{i,t}, x_{i,t+1}$

Figure 12: DeepAR decoder network

$\boldsymbol{h}_{i,t}$. The model is autoregressive and recurrent as it uses the previous output $\tilde{z}_{i,t}$ and state $\boldsymbol{h}_{i,t}$ as input, which potentially means that prediction errors at time t will negatively affect predictions at time $t > 1$.
The initial state $\boldsymbol{h}_{i,t_0-1}$ of the decoder shown in figure 12 is obtained using an encoder with the same architecture and weights that computes equation 3 for $t = 1, ..., t_0 - 1$. The forecasts $\tilde{\boldsymbol{z}}_{i,t_0:T}$ are obtained by sampling $\tilde{z}_{i,t} \sim l(\cdot|\theta(\boldsymbol{h}_{i,t}, \Theta))$, where $\theta(\boldsymbol{h}_{i,t}, \Theta))$ are the parameters (e.g. mean and variance) of the distribution fixed during training and are directly predicted by the decoder network.

The likelihood $l(z|\theta)$ determines the noise model and should match the statistical properties of the data: a Gaussian likelihood can be used for real-valued data, a beta likelihood for data in the unit interval, and a negative-binomial likelihood for positive count data. For example, the Gaussian likelihood is parametrized using its mean and standard deviation, i.e. $\theta = (\mu, \sigma)$ where μ is obtained with an affine function of the network output $\boldsymbol{h}_{i,t}$ and the standard deviation is obtained by applying an affine transformation followed by a softplus activation to ensure $\sigma > 0$, Therefore, each likelihood with parameters θ requires a mapping from the decoder state $\boldsymbol{h}_{i,t}$ to θ whose parameters are learned by the network.

Without any modification, in order to handle different scales the network should learn to scale the input to an appropriate range and then invert the scaling. As the network has a limited operating range and some datasets exhibit a power-law of scales (such as the Amazon dataset of [30]), this issue was addressed by scaling the input values (e.g. $\tilde{z}_{i,t}$ and $z_{i,t}$) using an item-dependent factor ν_i. Then, before drawing samples from the distribution, the output of the network (e.g. the mean μ of the Gaussian) is multiplied by the scale. The scaling factor ν is set to be the average value of the time series: $\nu_i = 1 + \frac{1}{t_0}\sum_{t=1}^{t_0} z_{i,t}$. Finally, rather than training the network choosing random time series from the dataset, the probability of choosing a time series is proportional to its scale factor ν_i: by non-uniformly sampling during training, imbalanced datasets with fewer large scale time series are used more effectively.

2.4.6 DeepState

DeepState [31] is a generative model that combines state space models [41] with deep learning. The idea is to use a latent state $\boldsymbol{l}_t \in \mathbb{R}^D$ to encode time series components such as level, trend, and seasonality patterns, and parametrize the linear state space model (SSM) by using a recurrent neural network (RNN) whose weights are learned jointly from multiple time series and covariates.
The main advantage of SSM is that the model is easily interpretable, but when used with traditional models such as ARIMA and Exponential Smoothing it results in an univariate model that still requires a lot of human work that cannot be easily recycled for other time series. DeepState solves this issue by using neural networks to learn a global model from multiple time series without making strong assumptions and reducing the human effort, and solving the common interpretability issue of neural networks by fusing them with SSMs.

The goal of DeepState is to produce probabilistic forecasts for each time series $i = 1, ..., N$ given the past:

$$p(\boldsymbol{z}_{i,T_i+1:T_i+\tau}|\boldsymbol{z}_{i,1:T_i},\boldsymbol{x}_{i,1:T_i+\tau};\Phi)$$

where $\boldsymbol{z}_{i,T_i+1:T_i+\tau}$ are the τ future values, $\boldsymbol{z}_{i,1:T_i}$ are the known past values, $\boldsymbol{x}_{i,1:T_i+\tau}$ are the covariates that must be known beforehand for $t = 1, ..., T$, and Φ is the set of learnable parameters of the model (i.e. the RNN).
DeepState makes the assumption that time series are independent of each other when conditioned on the associated covariates $\boldsymbol{x}_{i,1:T}$. Nevertheless, the model is still able to learn and share patterns across time series as Φ is shared (and learned) between all of them.

SSMs use a latent state $\boldsymbol{l}_t \in \mathbb{R}^L$ encoding time series components (level, trend, seasonality) that evolves over time with linear transitions at each time step t:

$$\boldsymbol{l}_t = F_t\boldsymbol{l}_{t-1} + \boldsymbol{g}_t\epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0,1)$$

where F_t is the transition matrix and $\boldsymbol{g}_t\epsilon_t$ is a random innovation component. The latent state can be inspected to check and potentially change the encoded trend and seasonality, and is also used to obtain predictions. For example, considering a linear Gaussian observation model:

$$z_t = y_t + \sigma_t\epsilon_t, \quad y_t = \boldsymbol{a}_t^T\boldsymbol{l}_{t-1} + b_t, \quad \epsilon_t \sim \mathcal{N}(0,1) \tag{4}$$

with the initial state $\boldsymbol{l}_0 \sim \mathcal{N}(\boldsymbol{\mu}_0, \text{diag}(\boldsymbol{\sigma}_0^2))$, $\boldsymbol{a}_t \in \mathbb{R}^L$, $\sigma_t \in \mathbb{R}_{>0}$, and $b_t \in \mathbb{R}$ varying over time.
Therefore, the state space model of *one* time series is fully described by $\Theta_t = (\boldsymbol{\mu}_0, \boldsymbol{\sigma}_0, \boldsymbol{F}_t, \boldsymbol{g}_t, \boldsymbol{a}_t, b_t, \sigma_t)$, $\forall t > 0$, differing from the classical settings where Θ doesn't change with time.
To obtain $\Theta_{i,t}$ for the time series i, the DeepState model learns a mapping Ψ from the covariates $\boldsymbol{x}_{i,1:T_i}$ to the parameters $\Theta_{i,t}$:

$$\Theta_{i,t} = \Psi(\boldsymbol{x}_{i,1:T}, \theta)$$

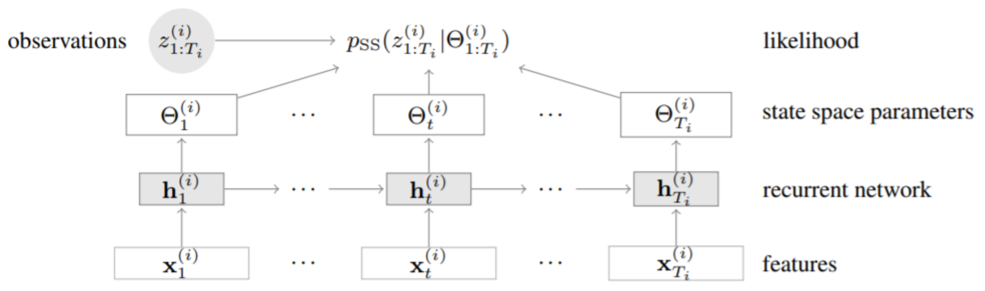


Figure 13: DeepState network

that is parametrized from a set of parameters θ learned jointly from the entire dataset of time series.
More precisely, the mapping Ψ is implemented by the RNN shown in figure 13 having a stacked architecture of LSTM cells. Its parameters θ are learned by maximizing the likelihood during training.

Finally, once the mapping is learned and given $\boldsymbol{x}_{i,1:T_i}$, the data $\boldsymbol{z}_{i,1:T_i}$ is distributed according to the marginal likelihood:

$$p(\boldsymbol{z}_{i,1:T_i}|\boldsymbol{x}_{i,1:T_i},\theta) = p_{SS}(\boldsymbol{z}_{i,1:T_i}|\Theta_{i,1:T_i})$$

$$= p(z_{i,1}|\Theta_{i,1})\prod_{t=2}^{T} p(z_{i,t}|z_{i,1:t-1},\Theta_{i,1:t})$$

$$= \int p(\boldsymbol{l}_0)[\prod_{t=1}^{T_i} p(z_{i,t}|\boldsymbol{l}_t)p(\boldsymbol{l}_t|\boldsymbol{l}_{t-1})]d\boldsymbol{l}_{0:T_i}$$

that is analytically tractable in the linear-Gaussian case.
To produce a forecast, the posterior of the last latent state $p(\boldsymbol{l}_T|z_{i,1:T_i})$ is computed using the observations $z_{i,1:T_i}$, then the RNN is fed with the covariates $x_{i,1:T_i+\tau}$ (see figure 14) while the transition equation is recursively applied, drawing Monte Carlo samples using equation 4.

2.4.7 Multi Quantile Recurrent Forecaster

Multi Quantile Recurrent Forecaster (MQCNN) [49] is a Sequence-to-Sequence RNN-based model capable of producing multi-horizon quantile forecasts.

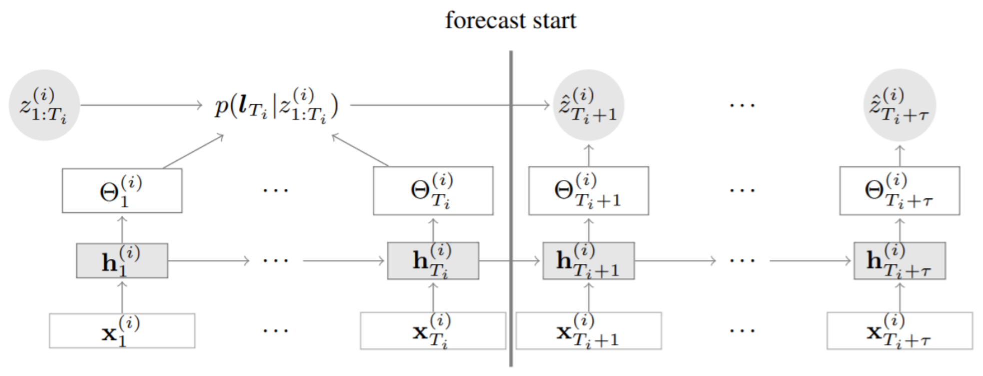

forecast start
$z^{(i)}_{1:T_i}$
$p(\boldsymbol{l}_{T_i}|z^{(i)}_{1:T_i})$
$\hat{z}^{(i)}_{T_i+1}$
$\hat{z}^{(i)}_{T_i+\tau}$
$\Theta^{(i)}_1$
$\Theta^{(i)}_{T_i}$
$\Theta^{(i)}_{T_i+1}$
$\Theta^{(i)}_{T_i+\tau}$
$\mathbf{h}^{(i)}_1$
$\mathbf{h}^{(i)}_{T_i}$
$\mathbf{h}^{(i)}_{T_i+1}$
$\mathbf{h}^{(i)}_{T_i+\tau}$
$\mathbf{x}^{(i)}_1$
$\mathbf{x}^{(i)}_{T_i}$
$\mathbf{x}^{(i)}_{T_i+1}$
$\mathbf{x}^{(i)}_{T_i+\tau}$

Figure 14: DeepState forecast illustration

[49] proposes a forking-sequences approach that improves the training stability and performance of encoder-decoder architectures by efficiently training on all time points where a forecast could be created. Furthermore, the model can be used with different encoders, but the best results were achieved using a CNN-based encoder.

To train a quantile regression model for a quantile $q \in [0, 1]$ the loss of a single forecasted value is given by:

$$L_q(y, \hat{y}) = q \max(0, y - \hat{y}) + (1 - q) \max(0, \hat{y} - y)$$

where by setting $q = 0.5$ the model will be trained to simply predict the median. Note that by predicting quantiles the model is robust since it doesn't make distributional assumptions (e.g. like DeepAr [30]).
Eventually, more quantiles can be considered such that the total loss is given by:

$$\textstyle\sum_{t \in T} \sum_{q \in Q} \sum_{k=1}^{K} L_q(y_{t+k}, \hat{y}^{(q)}_{t+k})$$

where T contains the forecast creation times, Q the quantiles, and K is the size of the horizon to forecast. Furthermore, different quantiles can be associated with different weights, which could be useful for tasks with an asymmetric cost for over and under-predicting.

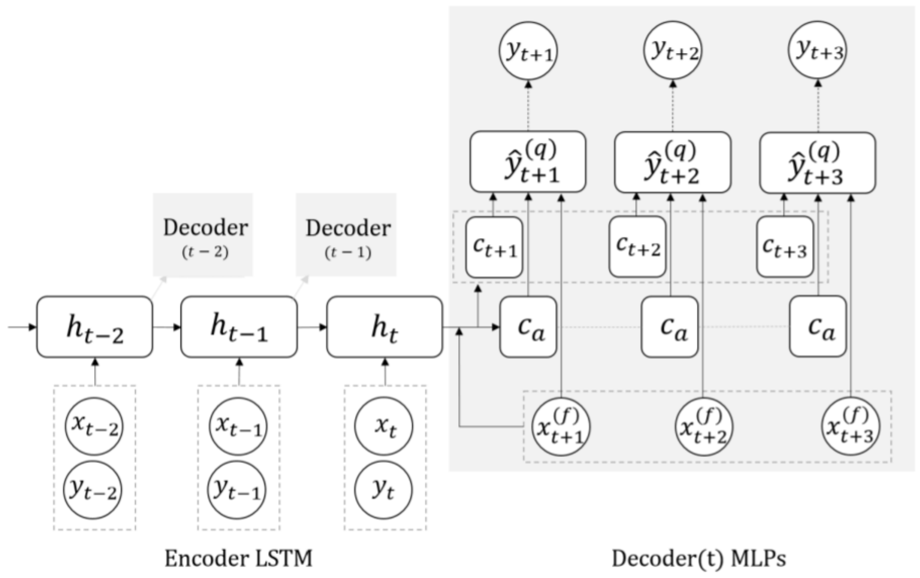


Figure 15: Multi-quantile recurrent forecaster architecture

The general architecture of a multi-quantile recurrent forecaster is shown in figure 15. The encoder is fed with the time series history producing hidden states h_t, then a global neural network summarizes the encoder output into an horizon-agnostic context c_a plus a horizon-specific context c_{t+k} for $k = 1, ..., K$ using the hidden state h_t and the future covariates $x_{t+1:t+K}$:

$$(c_{t+1,...,c_{t+K},c_a}) = m_G(h_t, x_{t+1:t+K})$$

where each context c_i can have arbitrary dimension. The idea behind this choice is that c_a should capture relevant information that is not time-sensitive, while c_{t+k} carries awareness of the temporal distance between the forecast creation time t and the specific horizon.
Then, these contexts are used by a local neural network to compute the quantiles of a specific horizon $t+k$ for each $k = 1, ..., K$ using the horizon-agnostic context and the horizon-specific context, plus the associated covariates:

$$(\hat{y}_{t+k}^{(q_1)}), ..., \hat{y}_{t+k}^{(q_Q)}) = m_L(c_{t+k}, c_a, x_{t+k})$$

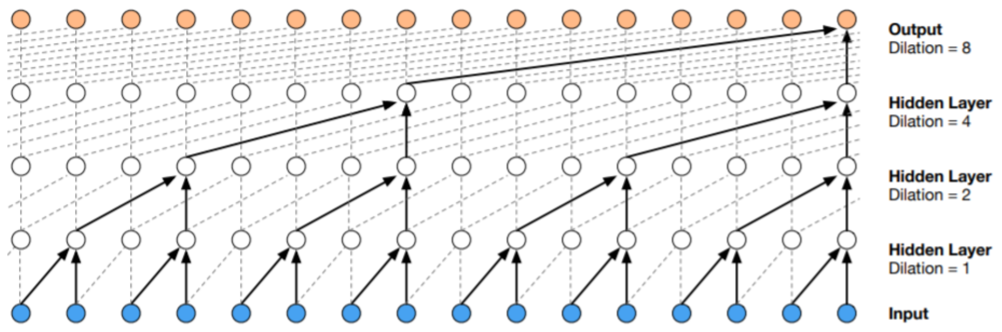


Figure 16: A stack of dilated causal convolutional layers

The local neural network implementing m_L has its parameters shared across all the horizons.

The motivation for replacing the standard RNN-based decoder is that the horizon-specific context should have already captured the flow of temporal information. Furthermore, by not feeding predictions recursively there is no error accumulation and following the forking-sequences training scheme proposed by [49] the training time is dramatically reduced while the process of updating the gradients is stabilized, leading to better forecasts with a reduced effort.

The encoder is not limited to be a simple LSTM-based RNN: [49] achieved the best results using a CNN-based encoder made of a stack of dilated causal convolutional layers, similarly to the work done by WaveNet [51].
Dilated causal convolutional layers, shown in figure 16, form long-term connections creating large receptive fields with just a few layers, thus preserving computational efficiency. The result is the so-called MQ-CNN model.

2.5 Workload forecasting

The ability of forecasting the workload of an IT system opens the possibility of proactively adapting the system according to the future demand and making smarter decisions, keeping the Quality of Service (QoS) high while reducing the infrastructure costs. This section lists some applications of workload forecasting and how it has been approached.

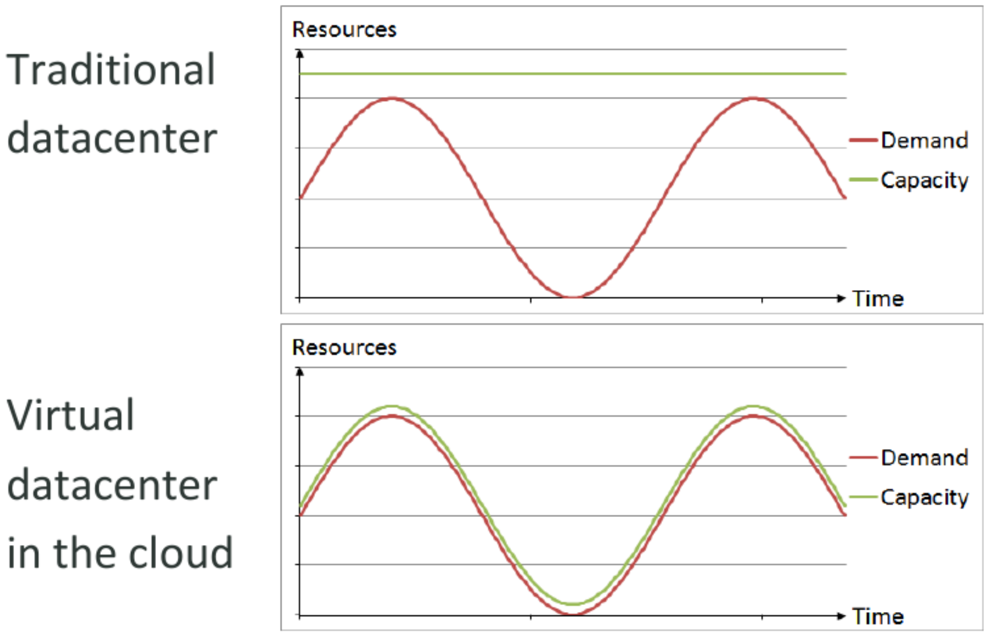


Figure 17: Cloud vs traditional computing.

Recent years have seen companies moving from self-hosted to cloud-hosted IT services, where a public provider is paid to lend on-demand computing power likewise utilities such as electricity, gas, and water. This paradigm, called cloud computing [52], enabled the possibility of flexibly adapting the capacity of a system according to the demand, potentially heavily reducing the infrastructure costs. Figure 17 highlights the difference of cloud computing with the traditional paradigm, where to avoid loosing customers the capacity of the system must be able to serve the highest peak of demand, causing an over-provisioning of resources when the demand is low.

A scalable system can acquire new resources in a matter of minutes, thus quickly reacting to changes in the demand. However, the demand a system is subject to can as quickly increase, making a reactive approach often inappropriate due to spikes in the demand that can cause disservices or even system crashes [53], leading to a loss of customers. In this context, a workload forecasting module that can make reliable forecasts about the upcoming workload allows to proactively adapt the system, reducing costs while pro-

viding high QoS. Such module must predict the workload by modeling time series with sub-hourly frequencies, representing metrics such as CPU usage and number of incoming requests.
To do so, [53] used ARIMA to predict the number of end-users' future requests to meet the QoS targets while reducing the resources utilization, focusing on specific request patterns that exhibit seasonal behavior. [54] used ARMA models to predict the upcoming workload and proactively autoscale the IT system under study, focusing on a small number of machines and leaving the exploration of the feasibility of their solution with modern workloads and large number of resources for future work. In order to deal with huge numbers of machines, with the goal of efficiently provision computing resources in the cloud, [55] adopted the approach of first grouping machines with high correlation and then making predictions about individual machine's workload based on the groups found on the previous step using Hidden Markov Model. [56] proposes the usage of LSTM-based neural networks to forecast the workload in large-scale computing centers, highlighting that training models on one-dimensional time series doesn't capture useful similarities across multiple time series.

Another task that makes use of workload forecasting is job scheduling. In the context of cloud service providers, running database backups while there are peaks of customer activity results in inevitable competition for resources and poor QoS. [57] proposed an automated solution to schedule backups during intervals of minimum activity comparing the forecasting models proposed by [37, 33, 36] in terms of accuracy and scalability. Interestingly, they discarded the ARIMA model due to its long execution time. Furthermore, by analyzing the typical customer activity patterns on PostgreSQL and MySQL servers, [57] discovered that the majority of the activity can be classified either as stable or as a daily or weekly pattern: less than 1% of the servers didn't follow either a daily or weekly pattern.

The application of novel forecasting techniques such as the ones based on neural network has still to be explored. Nevertheless, the flexibility of such models, especially when compared to ARIMA and Exponential Smoothing, is promising: potentially dealing with huge numbers of time series with minimum effort, while keeping the forecasting accuracy high, would make workload forecasting much more accessible to many companies, leading to better services and lower costs.

3 Proposed solution and approach

At the time of writing, Akamas [1] optimizes IT systems on a staging environment (a replica of the real system) using artificial workloads that have no impact on the user experience. The goal of this work is to extend the underlying tuner so it can be applied to the actual IT system running under the real workload, with the advantages of reducing the effort explained in section 2.2 and obtaining performance measurements directly from the "real" system. To do so, as explained in section 2.1.1, we need a workload characterization and a workload forecasting module.

The main challenge when tuning a system while it is serving its clients is to keep QoS levels high, and at the same time quickly finding good configurations for the system. Furthermore, we want Akamas to be easy to configure and apply in order to require the minimum amount of human work. Therefore, the requirements for the solution are to be autonomous and reliable.

The autonomous requirement translates to the need of a workload forecasting module that can train good models without having to inspect the time series composing the workload, and possibly without having inject domain knowledge about the system being optimized. The reliability requirement requires the models to be as accurate as possible. Furthermore, as the solution is running in real-time, it must be possible to train the forecasting models in a decent amount of time in order to incorporate new data, and the prediction queries must be satisfied in a matter of seconds.

As stated in section 2.1.1, we are interested in finding short time windows during which the workload will be stable, along with the average values the workload time series will assume.
Formally, given the forecast $\tilde{Y}_{t_1:t_2} = (\tilde{\boldsymbol{y}}^1_{t_1:t_2}, ..., \tilde{\boldsymbol{y}}^n_{t_1:t_2})$ at time t, where $\tilde{\boldsymbol{y}}^i_{t_1:t_2},\ i \in [1, n]$ is the forecast of the i-th time series composing the workload, we want to know if the window $\omega_{t_1:t_2}$ assuming values $\tilde{Y}_{t_1:t_2}$, starting at time $t_1 \geq t$

and ending at time $t_2 > t_1$ is stable:

$$s_\Theta(\tilde{Y}_{t_1:t_2}, Y_{t_0:t_1}) = \begin{cases} 1 & \text{if } \tilde{Y}_{t_1:t_2} \text{ is stable} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where s_Θ is a function with hyper-parameters Θ that marks whether a forecasted window is stable or not, given the historical values $Y_{t_0:t_1}$. The forecast $\tilde{Y}_{t_1:t_2}$ is also used by the tuner to suggest a workload-tailored configuration. To obtain such forecast, the forecasting module explained in section 3.2 was developed, while the component developed to implement the function s_Θ is detailed in section 3.3. Finally, the workload characterization module is presented in section 3.4.

3.1 Online Contextual Gaussian Process Tuner

Before starting the tuning process, made of iterations (or experiments) where a new configuration is repeatedly applied and evaluated, we must collect enough workload data in order to train the forecasting models and gather knowledge about the workload (see section 3.4 for details). As stated by [57], a significant number of workloads follow a daily or a weekly seasonality. Therefore, the data collection time lasts at least one week.

After the collection period has ended, the online tuning process of the IT system can start. The tuning process is made of experiments of duration up to $t_2 - t_1$, that is the length of the window $\omega_{t_1:t_2}$ on which a configuration is applied and its outcome is measured. However, it can happen that an experiment gets invalidated due to an early stop condition such as the violation of constraint (see section 2.1.1).

After a forecast is made, the experiment develops and the true workload reveals itself. When comparing the predicted with the actual workload, two (bad) cases can occur: the predicted average workload is different from the true average workload, and the stability prediction is not correct (e.g. predicted stable but was unstable). The latter case is further detailed by the four sub-cases shown in table 3.1. The most dangerous case occurs when

Case	Outcome
Predicted stable, revealed stable (TP)	Experiment opportunity taken
Predicted stable, revealed unstable (FP)	Experiment failed
Predicted unstable, revealed unstable (TN)	Experiment not available
Predicted unstable, revealed stable (FN)	Experiment opportunity lost

Table 2: Window stability outcomes. TP, FP, TN, FN stands for True Positive, False Positive, True Negative, and False negative respectively.

we predict that the workload will be stable but actually it won't (i.e. false positive case): in such case, the tuner may suggest a configuration that leads to low QoS, ruining user experience. However, if we miss a stable window (i.e. false negative case), we just lengthened the tuning process. In case of false positives we chose to discard the experiment as its evaluation requires us to measure the average performance of the configuration and an unstable workload may lead to unrealistic measurements (for example, a good configuration may be associated with a constrain violation due to a quick spike in the workload).
Similarly, we could face issues if the predicted and true windows are stable, but the actual average workload differs from the real average: the tuner would suggest a configuration tailored for a workload that is not the real one. In such cases, when adding the experiment to the knowledge of the tuner, we replace the average forecasted workload with the true average. By doing so, we may augment the knowledge base with a point that is not ideal, i.e. that doesn't maximize the acquisition function (see section 2.1). However, by replacing the associated workload, the point can still provide useful information to the tuner.

In summary, each experiment is made of the following steps:

1. Forecast the upcoming workload $\tilde{Y}_{t_1:t_2}$.
2. Apply $s_\Theta(\tilde{Y}_{t_1:t_2}, Y_{t_0:t_1})$ to find whether the upcoming workload $\tilde{Y}_{t_1:t_2}$ is stable.

3. If the upcoming workload is predicted to be unstable stop the experiment and go back to step 1, otherwise continue.

4. Ask the tuner a new configuration x given the average of the predicted workload $\tilde{Y}_{t_1:t_2}$ and the knowledge base (initially empty).

5. Apply the configuration x and monitor the state of the system.

6. While $t \in (t_1, t_2)$, check if any constraint violation occurred. If a violation occurred, check if it happened under an unstable workload by applying $s_\Theta(\tilde{Y}_{t:t_2}, Y_{t_0:t})$. If the workload was stable, add the violation caused by the configuration to the knowledge base, otherwise discard the experiment. Go back to step 1.

7. When $t = t_2$, check if the workload was actually stable applying $s_\Theta(Y_{t_1:t_2}, Y_{t_0:t_1})$. If it was stable: add the configuration-outcome pair to the knowledge base, otherwise discard it.
   Then, repeat from step 1.

It may happen that that forecasting module is called multiple times consecutively while the workload is unstable. To reduce the computational requirement, especially when deep learning models are used, forecasts are done for a window of length longer than the experiment, so that the same forecast can be used multiple times.

Before the tuner is queried for a new configuration, the workload characterization module groups the knowledge base by workload type (which are automatically detected) so that the performance of the system is normalized according to the related workload type (see section 2.1.1 and 3.4 for more details).

The reason for discarding violations that occur when the workload is unstable is that such violations may be caused by a difference between the predicted workload, that is used by the tuner to suggest a configuration, and the real workload, that may not coexist with that suggested configuration (e.g. an abnormal spike in the workload). In these cases we drop the experiment and start a new one. Furthermore, when a violation occurs, we quickly resort to the vendor (or baseline) configuration. An alternative approach could use the best configuration found so far for the current workload type.

Finally, when an experiment completes (i.e. at step 7) we store the evaluated point in the knowledge base using the true average workload rather than the predicted one.

It is important to note that both the forecasting and workload characterization modules are working together with a tuner that repeatedly applies new configurations to the system. Each new configuration will likely have an impact on some properties of the system being optimized, such as CPU and memory usage. If the workload being characterized and forecasted includes these properties, the forecasting models will face issues modeling time series with unpredictable changes caused by configuration changes, and the workload characterization module will not be able to objectively characterize workloads. Therefore, such system properties must not be part of the workload. In general, we characterize the workload using the number of users connected to the system and the read/write ratio, that are not affected by the configuration unless it has a catastrophic consequence on the system (e.g. a service no longer available). The number of requests gives an idea of the amount of work requested to the system, while the read/write ratio suggests the type of work.

After N iterations, the tuner will eventually converge to a good configuration for each type of workload. At that point, we can stop the tuner and use only the forecasting module to proactively apply such configurations.

Finally, note that the data collection time is exploited only by the forecasting module to train its models with a decent amount of history: the BO tuner doesn't make use of such time and it is therefore (busy) waiting until the end of the week. As a consequence, the first configuration suggestion will resort to the BO prior, which will likely lead to bad QoS levels. We could boost the suggestion of the first configurations by sampling the performance of the baseline configuration during the data collection period in order to initialize the knowledge base. However, by doing so the size of the knowledge base would quickly grow, slowing the actual tuning process (the complexity on the size of the knowledge base is $O(n^3)$). Therefore, such improvement requires us to summarize the knowledge base to reduce its size and is left as future work.

3.2 Forecasting module

As noted in section 2.4, different models may achieve different results for the same time series, depending on its properties (e.g. patterns, trends, cycles) and the available amount of data. Therefore, the forecasting module was developed such that it can wrap a different prediction model for each time series composing the workload. The models that have been included in the module are Prophet (section 2.4.3), DeepAR (section 2.4.5), DeepState (section 2.4.6), and MQCNN (section 2.4.7), plus two naive models that repeat the value of the previous day and the previous week. Besides Prophet and the naive models, the others can be used as multivariate models. The architecture is pictured in figure(TODO ADD ME or not?). Note that by using a well-defined model interface, it is very easy to integrate new models into the module.
Prophet is intended to be used when there is a small amount of data or when time series exhibit clear and strong seasonality pattern, while deep learning models should be used when more data is available.
If a time series exhibit a very strong daily or weekly pattern we can resort to the naive models, which are much lighter. In order to favor such lighter models we could penalize complexity, for example by using the Akaike information criterion.
Note that once the Prophet model is fitted it makes the same predictions independently from new data. On the other side, the deep learning models use the latest data each time a prediction is requested, allowing them to react to time series changes without re-fitting. However, the time required to fit a Prophet model is much less when compared to any neural network-based model in general.

The usage of the module is quite simple: as time advances, it will be called to add new data with a given frequency, eventually re-fitting the models to include new information. Meanwhile, the module can be called at any time to predict the upcoming workload.

In order to be configured, the *Forecaster* class accepts a JSON-formatted text that associates each time series with a model:

```
[
        {
                "name":"n_users",
                "model":"prophet",
                "interval":"5min",
                "memory":"30d",
                "group":"none"
        },
        {
                "name":"n_requests1",
                "model":"deepar",
                "interval":"5min",
                "memory":"30d",
                "group":"backend",
                "params":{
                        "train_epochs":30
                }
        },
        {
                "name":"n_requests2",
                "model":"deepar",
                "interval":"5min",
                "memory":"30d",
                "group":"backend",
                "params":{
                        "train_epochs":30
                }
        }
]
```

where the *group* property was used to build a DeepAR multivariate model on the time series *n_requests1* and *n_requests2*. The *params* property takes any model-specific parameter.
Finally, the *Forecaster* class provides utility methods to evaluate the accuracy of the predictions so that different models can be evaluated and the best one selected (see section 4.1).

3.3 Stable window finder

Given a forecast $\tilde{Y}_{t_1:t_2}$ we want to know if the represented workload is stable in order to effectively and safely apply a workload-tailored configuration suggested by the tuner. The stability function $s_\Theta(\tilde{Y}_{t_1:t_2}, Y_{t_0:t_1})$ is applied to each time series composing the workload independently, meaning that:

$$s_\Theta(\tilde{Y}_{t_1:t_2}, Y_{t_0:t_1}) = \wedge_{i=1}^{n} s_\Theta(\tilde{\boldsymbol{y}}^i_{t_1:t_2}, \boldsymbol{y}^i_{t_0:t_1})$$

where $\wedge_{i=1}$ is a *logical AND* operation, meaning that the workload is assumed to be stable if all the time series in the workload are independently stable. Θ is a hyper-parameter of the stability function, usually a threshold.

The function s_Θ is implemented by the module shown in picture (TODO ADD ME or not?). The reason for using the time series history as a parameter is that the evaluation of the stability of the upcoming values should take into consideration the past values, for example to know the range of values assumed by the time series. The following stability functions have been implemented:

- Coefficient of Variation (CV): a window is considered stable if its coefficient of variation Θ doesn't exceed a threshold. This method doesn't make use of past values.

$$s_\Theta(\tilde{\boldsymbol{y}}_{t_1:t_2}, \boldsymbol{y}_{t_0:t_1}) = \begin{cases} 1 & \text{if } \frac{\sigma(\tilde{\boldsymbol{y}}_{t_1:t_2})}{\mu(\tilde{\boldsymbol{y}}_{t_1:t_2})} > \Theta \\ 0 & \text{otherwise} \end{cases}$$

- Min-Max: let $\delta(\boldsymbol{x}) = (\max \boldsymbol{x} - \min \boldsymbol{x})$ be the width of the range of values assumed by $\boldsymbol{x}$. Then, a window is considered stable if its values are in a range with size that is below a threshold Θ times the size of the range of values assumed in the whole history of the time series.

$$s_\Theta(\tilde{\boldsymbol{y}}_{t_1:t_2}, \boldsymbol{y}_{t_0:t_1}) = \begin{cases} 1 & \text{if } \delta(\tilde{\boldsymbol{y}}_{t_1:t_2}) \leq \Theta \cdot \delta(\boldsymbol{y}_{t_0:t_1}) \\ 0 & \text{otherwise} \end{cases}$$

- Normal: normalize the window values using the mean and standard deviation of $\boldsymbol{y}_{t_0:t_1}$. Let $\mu_N(\tilde{\boldsymbol{y}}_{t_1:t_2})$ be the (normalized) mean. Then the window is considered stable if all its (normalized) values y are such that: $y \in [\mu_N(\tilde{\boldsymbol{y}}_{t_1:t_2}) - \Theta, \mu_N(\tilde{\boldsymbol{y}}_{t_1:t_2}) + \Theta]$

Of these methods, the minimum-maximum based one is the most intuitive, as Θ sets the allowed percentage of movement from the historical values.

Note that once we have the true workload, we can check whether a window was actually stable or not by running $s_\Theta(\boldsymbol{y}_{t_1:t_2}, \boldsymbol{y}_{t_0:t_1})$. This is extremely important because it enables us to invalidate a tuning experiment if the workload revealed to be unstable. Furthermore, it allows us to easily evaluate the stability algorithm or tune its parameters Θ.

Finally, note that we are using a wide variety of forecasting models with different properties. For example, Prophet (section 2.4.3) doesn't model noise and therefore its predictions are flatter when compared to the real time series or other models' predictions. This means that by using the same threshold for predicting if a window will be stable and then checking if the window was actually stable eventually leads to misleading outcomes, even if the forecast is accurate. Therefore, we use two different thresholds $\tilde{\Theta}$ and Θ: the former for the prediction, i.e. $s_{\tilde{\Theta}}(\tilde{\boldsymbol{y}}_{t_1:t_2}, \boldsymbol{y}_{t_0:t_1})$, and the latter for the posterior evaluation $s_\Theta(\boldsymbol{y}_{t_1:t_2}, \boldsymbol{y}_{t_0:t_1})$.

In order to set the thresholds effectively, a human operator must first of all vary Θ to see which windows would be considered stable on the true workload time series (see figure 18). Note that it is not trivial to find an optimal $\Theta^\star$ that fits all scenarios because its effectiveness depends on the properties of the workload received by the system, such as noise. To find $\tilde{\Theta}$,

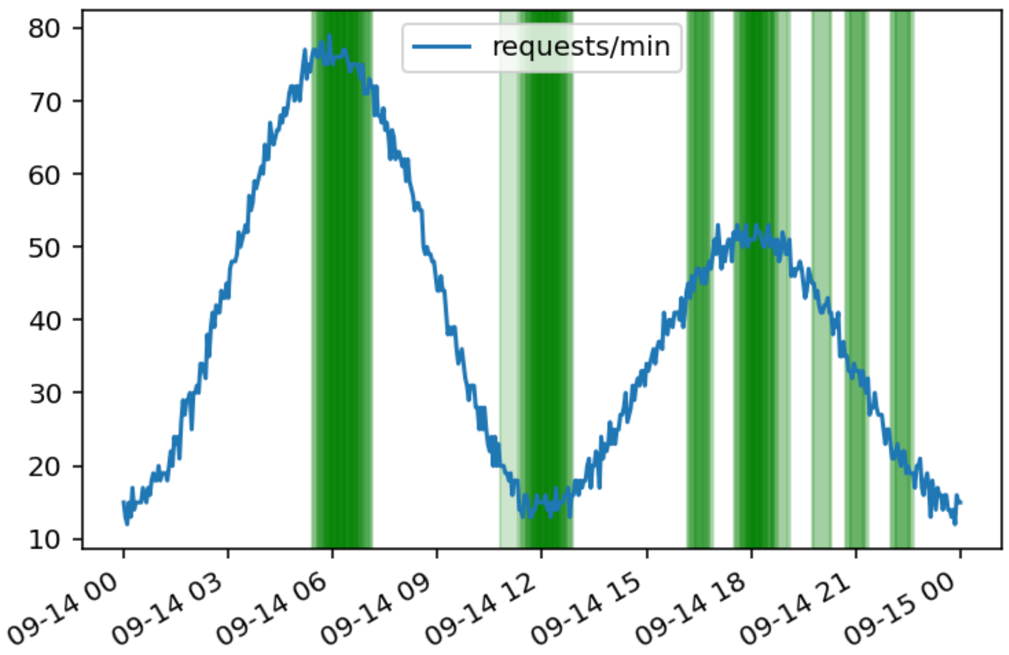


Figure 18: Example of stable windows detected by Min-Max algorithm using a threshold of 6%.

we need to train the forecasting model, produce forecasts, and then find a value that matches the outcome of the previous step (both stable and unstable windows). Note that this step could be automated by finding $\tilde{\Theta}$ that maximizes the number of stable and unstable windows matches resulting from $s_\Theta(\boldsymbol{y}_{t_1:t_2}, \boldsymbol{y}_{t_0:t_1})$ and $s_{\tilde{\Theta}}(\tilde{\boldsymbol{y}}_{t_1:t_2}, \boldsymbol{y}_{t_0:t_1})$:

$$\tilde{\Theta} = \underset{\tilde{\Theta}}{\text{argmax}} \frac{n_{TP,\tilde{\Theta}}}{n_{TP,\tilde{\Theta}} + w \cdot n_{FP,\tilde{\Theta}}}$$

where $n_{TP,\tilde{\Theta}}$ is the number of true positives, i.e. the number of predicted stable windows that were actually stable, $n_{FP,\tilde{\Theta}}$ is the number of false positives (see table 3.1 for more details), and w is a weight that allows to prioritize the absence of false positives over true positive.

3.4 Workload characterization module

Workload characterization is used to normalize the outcome of each configuration according to the workload type, which is required by contextual Bayesian optimization (section 2.1.1). We are interested in characterizing the workload without any human intervention: for this reason, the module uses clustering-based methods.
The clustering methods that have been chosen are k-means, mean shift, Gaussian Mixture Models (GMM), and OPTICS (see section 2.3).

Since k-means requires the number of clusters k to be given as input, the quality of clustering is evaluated for each size $k \in K, K = [2, \max(3, \log n)]$ where n is the number of workload points. The evaluation is performed computing the Silhouette score for each k, and selecting the value that leads to the highest score:

$$k = \operatorname*{argmax}_{k \in K} S(W, l_k)$$

where S is the Silhouette score, W is the set of workload points, and l_k are the labels obtained by applying k-means to find k clusters.

Mean shift doesn't require to set the number of clusters beforehand, but it must be provided with the size of the bandwidth used in the RBF kernel. The bandwidth is estimated using k-nearest neighbor on a down-sampled workload dataset to reduce the computation time.

GMM must be provided with the number of components (i.e. clusters) to look for and the covariance type. These parameters are chosen by maximizing the Bayesian Information Criterion (BIC):

$$(n, cv) = \operatorname*{argmax}_{n \in N, cv \in CV} BIC(C_{n,cv})$$

where n is the number of components, cv is the covariance type, and $C_{n,cv}$ is the clustering obtained using GMM with n and cv. Similarly to k-means $N = [2, \max(3, \log n)]$, while CV is the set of available covariance types, e.g *spherical*, and *diagonal*.

Finally, OPTICS clustering is performed using the Euclidean distance and with different values of ξ, that controls a cluster boundary. Similarly to k-means, the best value of ξ is chosen by maximizing the Silhouette score:

$$\xi = \operatorname*{argmax}_{\xi \in \Xi} S(W, l_\xi)$$

where Ξ is the set of possible ξ values, and l_ξ are the labels obtained by applying OPTICS on the workload dataset W using ξ. The workload points marked as outliers are assigned to a dedicated cluster containing just one point.

In all cases, since the properties composing the workload may have different scales (e.g. number of users and read/write ratio), they are all scaled in the $[0, 1]$ range using a min-max scaler. Furthermore, with all methods except OPTICS, the size of the input dataset W is limited by randomly sampling N points from W so that the computation effort required by the clustering methods is limited.

Finally, the dataset W is initialized together with the forecasting module: the workload points seen during the initialization period (e.g. the first week) are included so that the tuner is provided with meaningful workload groups since the beginning of the optimization process.

4 Experimental setup

The proposed solution has the goal of extending the existing tuner [1] to work in an online manner. Therefore, we are interested in how quickly we are able to find a good configuration while avoiding bad ones. Nevertheless, the effectiveness of the tuning depends on two key factors: the ability of selecting as many stable workload windows as possible, that increases the number of experiments and therefore how quickly we can optimize the objective function, while avoiding unstable workload windows that are likely to lead to failures or low QoS levels. This selection ability, along with the quality of the configuration proposed by the tuner, depends on the forecasting accuracy.

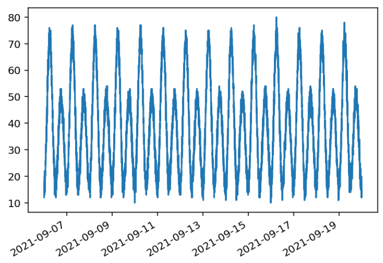
Therefore, we evaluate the forecasting module and the window selection algorithm independently from the tuner (section 4.1 and 4.2). The workload characterization module, that is based on clustering algorithms, is evaluated qualitatively. The overall solution is evaluated by using DBMS models (section 4.3) that map the configuration space to a wide variety of performance metric (according to the perceived workload), allowing us to run reproducible tests on a local machine.

To evaluate each component we chose a set of four time series to make the experiments reproducible (figure 19). Such time series represent the number of users connected to the system being optimized, i.e. the DBMS models. Two time series are synthetically generated by creating a daily and a weekly pattern. The daily pattern time series has three variants with increasing noise to make a sensibility analysis (figure 20). The remaining two time series are obtained from real data: the number of taxi requests in the city of New York and the number of requests to a bank system. Note that all time series have been mapped to the $[10, 80]$ range to be compatible with the DBMS models.
Furthermore, besides the number of requests, the workload can be affected by the behavior of the users at a certain time (e.g. are the users just reading data from the DBMS or are they inserting data?). We model this property of the workload by using three types of user behavior: read-intensive, write-intensive, and balanced.

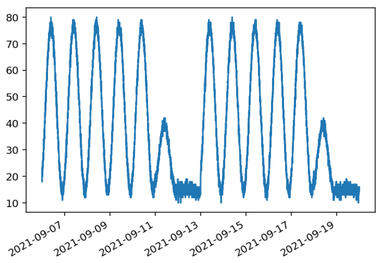
Finally, we are interested in the performance of the developed solution when the amount of time series data provided to the forecaster covers less than one week. By exploring such case when using time series with weekly seasonality we can also understand how the system reacts to patterns that are not present in the historical data (e.g. the reduced number of requests during the weekend when the forecaster has data up to Thursday).

4.1 Workload Forecasting

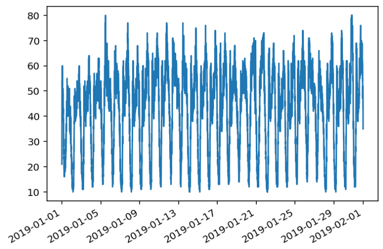
To evaluate the forecasting accuracy we chose two metrics: the Mean Absolute Percentage Error (MAPE) and the Root Mean Squared Error (RMSE). Remembering that we are repeatedly forecasting short-term windows while



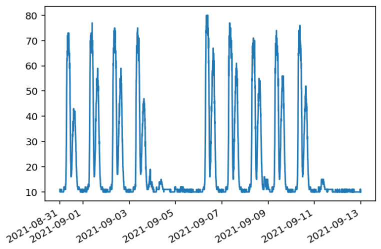
(a) Daily pattern



(b) Weekly pattern

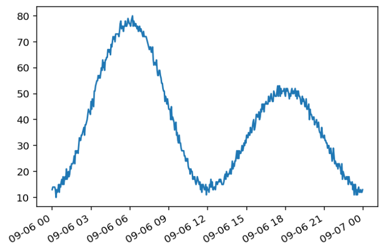


(c) Taxi requests

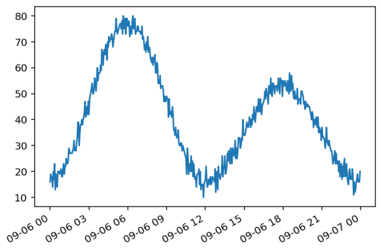


(d) Bank requests

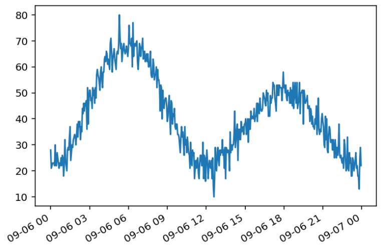
Figure 19: Workload time series.



(a) Daily 0.02 stddev noise



(b) Daily 0.04 stddev noise



(c) Daily 0.08 stddev noise

Figure 20: Zoomed daily pattern with increasing noise.

new workload data is incoming, we are interested in the overall errors up to the latest measurement plus how wrong the forecaster is in a specific moment.

Formally, when evaluating the accuracy of a time series starting at time t_0 and we have data up to time t_1, for each forecasted window $\omega_{t:t+\delta}$ where δ is the length of the short term forecast, we compute the error of the associated forecast $\tilde{\boldsymbol{y}}_{t:t+\delta}$ using the true values $\boldsymbol{y}_{t:t+\delta}$, for $t = t_0, ..., t_1 - \delta$. Then, we merge the forecasts and compute the error up to time $t_1 - \delta$. As we receive new time series data over time (i.e. t_1 increases over time), the latter is called incremental MAPE or RMSE.
Figure 21 shows a time series with naive forecasts, the per-forecast error and the incremental error evolving over time when the tuning windows have length 1 hour.

The error of a forecasting model in a specific moment can be used to inspect the performance of a single model, eventually automatically triggering a re-fitting process when the error exceeds a threshold. For example, picture (b) in figure 21 clearly shows a huge forecasting error on date 2019-01-28 given by configuration zero. On the other side, the incremental error can be used to compare different forecasting models and how the errors change after a model has been re-fitted (e.g. new data has been included). The latest value of the incremental error is of particular interest, as it represents the up-to-date overall performance of a model.

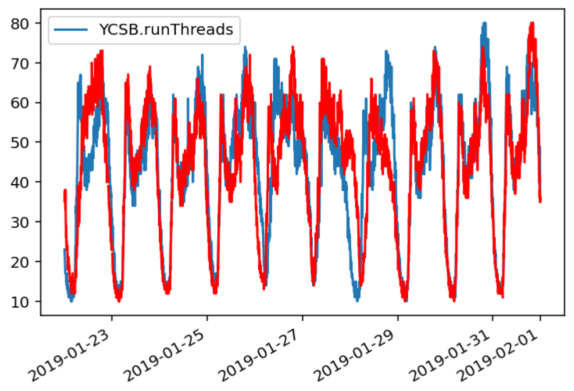
The MAPE has the following formula:

$$\text{MAPE}(\boldsymbol{y}_{t:t+m}, \tilde{\boldsymbol{y}}_{t:t+m}) = \tfrac{100}{m} \textstyle\sum_{i=t}^{t+m} |\tfrac{\boldsymbol{y}_{t:t+m} - \tilde{\boldsymbol{y}}_{t:t+m}}{\boldsymbol{y}_{t:t+m}}|$$

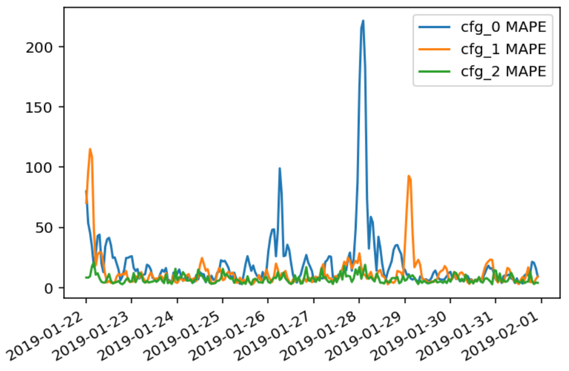
and the RMSE:

$$\text{RMSE}(\boldsymbol{y}_{t:t+m}, \tilde{\boldsymbol{y}}_{t:t+m}) = \sqrt{\tfrac{\sum_{i=t}^{t+m} (y_i - \tilde{y}_i)^2}{m}}$$

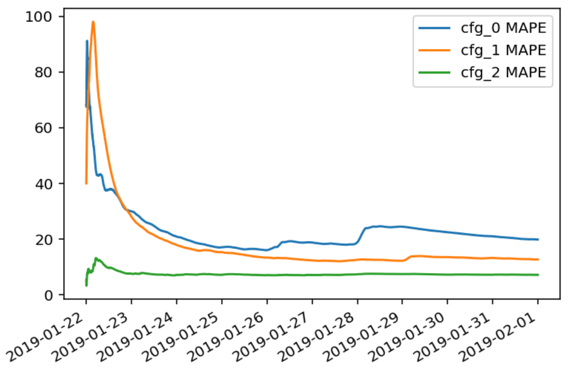
The two metrics may look redundant, but since the RMSE gives more importance to the highest errors, by considering the RMSE over the MAPE we are actually increasing sensitivity to outliers. Furthermore, using RMSE



(a) Naive forecasts (red) based on previous day value.



(b) Per-forecast MAPE.



(c) Incremental MAPE.

Figure 21: Per-forecast and incremental MAPE example of three models. The forecast shown in picture (a) is given by configuration 0.

we seek to be correct on average, while in contrast the MAPE targets the median.
Finally, note that the MAPE is easier to understand as it express the error as a percentage that is independent from the scale of the time series. Therefore, it is useful to compare the performance of a model on different time series. Nevertheless, MAPE cannot be used when the values are too close to zero.

4.2 Stable workload finder

The stable workload finder algorithm uses the forecast of the upcoming workload and its historical data to predict if a time window in the upcoming future will be stable or not. Therefore, it acts as a binary classifier that should detect a decent amount of stable windows while avoiding unstable ones. As mentioned in section 3.3, we give more importance to avoiding false positives (see table 3.1) as they potentially lead to bad QoS levels.

To evaluate such classifier, we consider its precision and recall. Precision is defined as:

$$\text{Precision} = \frac{True\,Positive}{True\,Positive + False\,Positive}$$

and Recall as:

$$\text{Recall} = \frac{True\,Positive}{True\,Positive + False\,Negative}$$

Simply put, Precision measures how many of the windows that were marked stable were actually stable, while Recall measures how many (true) stable windows were found by the algorithm. Therefore, we give more importance to the Precision score as it measures how many potentially dangerous experiments the tuner performed. On the other side, a good Recall value means that the tuner exploited as many windows as possible eventually leading to faster optimal convergence, but this shouldn't come at the cost of QoS.

To balance Recall and Precision we use the F1 score:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

4.3 Online Contextual Gaussian Process Tuner

As mentioned before, the proposed solution is evaluated using DBMS models, namely MongoDB and Cassandra. The models are trained such that they can map the configuration space, specific to each DBMS, to a wide variety of performance metrics [1], such as throughput and memory usage. By doing so, the experiments are easily reproducible.
Furthermore, by using database models we know the optimal configuration and we can see how far the tuner is from the real optimum. To do so, we compute the Normalized Performance Improvement at iteration i:

$$NPI(i) = \frac{y_0 - y_i}{y_0 - y^\star} = \frac{f(\boldsymbol{x}_0\boldsymbol{w}_i) - f(\boldsymbol{x}_i, \boldsymbol{w}_i)}{f(\boldsymbol{x}_0\boldsymbol{w}_i) - f(\boldsymbol{x}^\star_{\boldsymbol{w}_i}, \boldsymbol{w}_i)}$$

where $\boldsymbol{x}_0$ is the vendor (or baseline) configuration, $\boldsymbol{x}_i$ is the configuration being evaluated at iteration i, and $\boldsymbol{x}^\star_{\boldsymbol{w}_i}$ is the optimal configuration for the workload observed at iteration i. Therefore, an NPI of zero means that we have the same performance of the vendor configuration, and an NPI of one means that we found the global optimum.

To evaluate the entire tuning process up to iteration i we use the Cumulative Reward (CR):

$$CR(i) = \sum_{j=0}^{i} NPI(j)$$

that has a constant unitary slope if the tuner is optimal, is equal to zero if the baseline is repeatedly applied, and has no lower bounds for bad configurations.
By using normalized performance metrics we can quantitatively compare and evaluate tuners independently from the workload, taking into consideration the number of iterations required by the tuner to find a good configuration.

Furthermore, we consider the cumulative number of failures occurred up to iteration i:

$$\text{Failures(i)} = \textstyle\sum_{j=0}^{i} F(j)$$

where $F(j)$ is one if a failure occurred at iteration j, zero otherwise. The constraint we are imposing to the tuner is to keep the latency (i.e. the response time) below some QoS target depending on the tuning scenario difficulty, e.g. 10 ms.

The objective function we are trying to minimize is the cache size of the database models, which are MongoDB and Cassandra.

5 Results

6 Conclusions

7 Future work

References

[1] S. Cereda, S. Valladares, P. Cremonesi, and S. Doni, “Cgptuner: A contextual gaussian process bandit approach for the automatic tuning of it configurations under varying workload conditions,” *Proc. VLDB Endow.*, vol. 14, p. 1401–1413, apr 2021.

[2] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, “Automatic database management system tuning through large-scale machine learning,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD ’17, pp. 1009–1024, 2017.

[3] D. V. Aken, D. Yang, S. Brillard, A. Fiorino, B. Zhang, C. Billian, and A. Pavlo, “An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems,” *Proc. VLDB Endow.*, vol. 14, no. 7, pp. 1241–1253, 2021.

[4] P. Jamshidi, M. Velez, C. Kästner, and N. Siegmund, “Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, (New York, NY, USA), p. 71–82, Association for Computing Machinery, 2018.

[5] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, “Taking the human out of the loop: A review of bayesian optimization,” *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2016.

[6] A. Krause and C. Ong, “Contextual gaussian process bandit optimization,” in *Advances in Neural Information Processing Systems* (J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, eds.), vol. 24, Curran Associates, Inc., 2011.

[7] M. C. Calzarossa, L. Massari, and D. Tessera, “Workload characterization: A survey revisited,” *ACM Comput. Surv.*, vol. 48, feb 2016.

[8] W. Kirch, ed., *Pearson's Correlation Coefficient*, pp. 1090–1091. Dordrecht: Springer Netherlands, 2008.

[9] R. Xu and D. Wunsch, "Survey of clustering algorithms," *IEEE Transactions on Neural Networks*, vol. 16, no. 3, pp. 645–678, 2005.

[10] A. Maćkiewicz and W. Ratajczak, "Principal components analysis (pca)," *Computers & Geosciences*, vol. 19, no. 3, pp. 303–342, 1993.

[11] A. Mahanti, N. Carlsson, A. Mahanti, M. Arlitt, and C. Williamson, "A tale of the tails: Power-laws in internet measurements," *IEEE Network*, vol. 27, no. 1, pp. 59–64, 2013.

[12] P. Hansen and B. Jaumard, "Cluster analysis and mathematical programming," *Math. Program.*, vol. 79, pp. 191–215, 10 1997.

[13] M.-S. Yang, "A survey of fuzzy clustering," *Mathematical and Computer Modelling*, vol. 18, no. 11, pp. 1–16, 1993.

[14] S. C. Johnson, "Hierarchical clustering schemes," *Psychometrika*, vol. 32, pp. 241–254, 1967.

[15] J. Kleinberg, "An impossibility theorem for clustering," in *Proceedings of the 15th International Conference on Neural Information Processing Systems*, NIPS'02, (Cambridge, MA, USA), p. 463–470, MIT Press, 2002.

[16] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," tech. rep., Stanford, 2006.

[17] Y. Cheng, "Mean shift, mode seeking, and clustering," *IEEE transactions on pattern analysis and machine intelligence*, vol. 17, no. 8, pp. 790–799, 1995.

[18] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure," in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*,

SIGMOD '99, (New York, NY, USA), p. 49–60, Association for Computing Machinery, 1999.

[19] G. J. McLachlan and K. E. Basford, *Mixture models: Inference and applications to clustering*, vol. 38. M. Dekker New York, 1988.

[20] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, 1987.

[21] A. A. Neath and J. E. Cavanaugh, "The bayesian information criterion: background, derivation, and applications," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 4, no. 2, pp. 199–203, 2012.

[22] G. Mahalakshmi, S. Sridevi, and S. Rajaram, "A survey on forecasting of time series data," in *2016 International Conference on Computing Technologies and Intelligent Data Engineering (ICCTIDE'16)*, pp. 1–8, 2016.

[23] C. C. Holt, "Forecasting seasonals and trends by exponentially weighted moving averages," *International Journal of Forecasting*, vol. 20, no. 1, pp. 5–10, 2004.

[24] G. E. P. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control.* 5th ed., 2015.

[25] J. G. De Gooijer and R. J. Hyndman, "25 years of time series forecasting," *International Journal of Forecasting*, vol. 22, no. 3, pp. 443–473, 2006. Twenty five years of forecasting.

[26] B. Lim and S. Zohren, "Time-series forecasting with deep learning: a survey," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 379, no. 2194, p. 20200209, 2021.

[27] A. Sherstinsky, "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network," *CoRR*, vol. abs/1808.03314, 2018.

[28] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," 2014.

[29] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *CoRR*, vol. abs/1406.1078, 2014.

[30] V. Flunkert, D. Salinas, and J. Gasthaus, "Deepar: Probabilistic forecasting with autoregressive recurrent networks," *CoRR*, vol. abs/1704.04110, 2017.

[31] S. S. Rangapuram, M. W. Seeger, J. Gasthaus, L. Stella, Y. Wang, and T. Januschowski, "Deep state space models for time series forecasting," in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, Curran Associates, Inc., 2018.

[32] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, "The m4 competition: Results, findings, conclusion and way forward," *International Journal of Forecasting*, vol. 34, no. 4, pp. 802–808, 2018.

[33] A. Alexandrov, K. Benidis, M. Bohlke-Schneider, V. Flunkert, J. Gasthaus, T. Januschowski, D. C. Maddix, S. S. Rangapuram, D. Salinas, J. Schulz, L. Stella, A. C. Türkmen, and Y. Wang, "Gluonts: Probabilistic time series models in python," *CoRR*, vol. abs/1906.05264, 2019.

[34] S. Smyl, "A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting," *International Journal of Forecasting*, vol. 36, no. 1, pp. 75–85, 2020. M4 Competition.

[35] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, "The m5 accuracy competition: Results, findings and conclusions," 10 2020.

[36] S. Taylor and B. Letham, "Forecasting at scale," *The American Statistician*, vol. 72, 09 2017.

[37] "Ssaforecaster." https://docs.microsoft.com/en-us/python/api/nimbusml/nimbusml.timeseries.ssaforecaster?view=nimbusml-py-latest.

[38] R. Hyndman and G. Athanasopoulos, *Forecasting: Principles and Practice.* Australia: OTexts, 2nd ed., 2018.

[39] C. R. B., C. W. S., M. J. E., and T. I. J., "Stl: A seasonal-trend decomposition procedure based on loess,"

[40] A. Y. Ng and M. I. Jordan, "On discriminative vs. generative classifiers: a comparison of logistic regression and naive bayes," *Advances in Neural Information Processing Systems*, no. 14, pp. 841–848, 2002.

[41] R. Hyndman, A. Koehler, K. Ord, and R. Snyder, *Forecasting with exponential smoothing. The state space approach.* 01 2008.

[42] S. MAKRIDAKIS and M. HIBON, "Arma models and the box–jenkins methodology," *Journal of Forecasting*, vol. 16, no. 3, pp. 147–163, 1997.

[43] R. J. Hyndman and Y. Khandakar, "Automatic time series forecasting: The forecast package for r," *Journal of Statistical Software*, vol. 27, no. 3, p. 1–22, 2008.

[44] T. Hastie and R. Tibshirani, "Generalized additive models: Some applications," *Journal of the American Statistical Association*, vol. 82, no. 398, pp. 371–386, 1987.

[45] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, "A limited memory algorithm for bound constrained optimization," *SIAM Journal on Scientific Computing*, vol. 16, no. 5, pp. 1190–1208, 1995.

[46] H. Hewamalage, C. Bergmeir, and K. Bandara, "Recurrent neural networks for time series forecasting: Current status and future directions," *CoRR*, vol. abs/1909.00590, 2019.

[47] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990.

[48] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 6, p. 107–116, apr 1998.

[49] R. Wen, K. Torkkola, B. Narayanaswamy, and D. Madeka, "A multi-horizon quantile recurrent forecaster," 2018.

[50] R. DiPietro, C. Rupprecht, N. Navab, and G. D. Hager, "Analyzing and exploiting narx recurrent neural networks for long-term dependencies," 2018.

[51] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *CoRR*, vol. abs/1609.03499, 2016.

[52] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, p. 50–58, apr 2010.

[53] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using arima model and its impact on cloud applications' qos," *IEEE Transactions on Cloud Computing*, vol. 3, no. 4, pp. 449–458, 2015.

[54] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *2011 IEEE 4th International Conference on Cloud Computing*, pp. 500–507, 2011.

[55] A. Khan, X. Yan, S. Tao, and N. Anerousis, "Workload characterization and prediction in the cloud: A multiple time series approach," in *2012 IEEE Network Operations and Management Symposium*, pp. 1287–1294, 2012.

[56] X. Tang, "Large-scale computing systems workload prediction using parallel improved lstm neural network," *IEEE Access*, vol. 7, pp. 40525–40533, 2019.

[57] O. Poppe, T. Amuneke, D. Banda, A. De, A. Green, M. Knoertzer, E. Nosakhare, K. Rajendran, D. Shankargouda, M. Wang, A. Au, C. Curino, Q. Guo, A. Jindal, A. Kalhan, M. Oslake, S. Parchani, V. Ramani, R. Sellappan, S. Sen, S. Shrotri, S. Srinivasan, P. Xia, S. Xu, A. Yang, and Y. Zhu, "Seagull: An infrastructure for load prediction and optimized resource allocation," *CoRR*, vol. abs/2009.12922, 2020.