Received 14 December 2023, accepted 28 December 2023, date of publication 12 January 2024, date of current version 22 January 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3353201

APPLIED RESEARCH

Probabilistic Temporal Fusion Transformers for Large-Scale KPI Anomaly Detection

HAORAN LUO[1], (Member, IEEE), YONGKUN ZHENG[1], KANG CHEN[1], (Member, IEEE), AND SHUO ZHAO[2]
[1]Research Institute of China Telecom Company Ltd., Guangzhou 510630, China
[2]China Telecom Corporation Ltd., Xicheng, Beijing 100033, China

Corresponding author: Haoran Luo (luohr1@chinatelecom.cn)

This work was supported by China Telecom Company.

ABSTRACT This paper introduces a new generic and scalable framework for large-scale time series prediction and unsupervised anomaly detection. The most common approach of state-of-the-art time series anomaly detection techniques, which are mostly based on neural networks, is to train a network per time series. However, a typical modern microservice system consists of hundreds of active nodes/instances. To monitor the performance of such a system, we often need to keep track of thousands of time series describing different aspects of the system, including CPU usage, call latency, and workloads. We introduce a new methodology for grouping metrics that share the same type, predicting hundreds of metrics concurrently with a single neural network model with shared parameters. The model also integrates the probabilistic representations and Temporal Fusion Transformers for better performance. In a real-world dataset, our proposed model achieved up to 50% improvement in terms of MSE.

INDEX TERMS Deep learning, time series, anomaly diagnosis, microservice systems.

I. INTRODUCTION

Modern microservice systems consist of thousands of services, grouped by hundreds of subsystems. Generally, each service runs on a different container, and those containers can be dynamically created or destroyed according to some scaling configurations. It is challenging for the operator to detect operational issues and perform swift troubleshooting in such a complex environment.

During runtime, an operator can extract the metrics generated by nodes and services, such as CPU/Mem usage, Network I/O and the number of requests per second. Combining the observations from different timestamps forms some time series; by analyzing such series, an operator can decide whether the system is in an abnormal state; predicting the future values of time series also helps the operator discover the hidden risks before it actually break the system.

Traditional time series analysis methods require intensive human effort. To analyze the performance of a certain service, an operator must observe the time series generated by that service, then set the thresholds manually according to his/her experience. However, in modern microservice systems, the great quantity of pods and containers makes the manual method unacceptable. Operators require an unsupervised method, which could automatically extract proper thresholds from previous time series with minimal human inference.

A rich body of literature investigates unsupervised time-series anomaly detection tasks. Xu et.al. proposed a model Donut [1] based on variational auto-encoders (VAE), which achieved 0.9 F-score in an unsupervised training setup. Ma et. al. proposed an approach based on a one-hot support vector machine to identify unforeseen or abnormal sections in a time series. Other unsupervised methods, such as clustering, are also being actively researched according to [2].

However, as the size of modern microservice system growing larger, the above methods reached a performance bottleneck. For example, training a single VAE or clustering model is not trivial; training different models (especially neural networks) for hundreds of sequences and predict them simultaneously becomes unacceptable in modern production settings. Therefore, we try to find a universal model for the same type of metrics in a microservice system.

The model should have the ability to effectively distinguish the ‘‘static’’ parts in different time series, and itself should

The associate editor coordinating the review of this manuscript and approving it for publication was Prakasam Periasamy.

be complex enough to hold time series with different characteristics. A competitive model is Temporal Fusion Transformer(TFT) [3], which integrates static info and LSTM layers into transformer structure, and combines them via residual layers. Based on the idea of TFT, we tested and optimized it according to the requirements in production environment of microservice system.

The contributions of this paper can be summarized as follows.

- We show that Temporal Fusion Transformers can be the state-of-the-art model for unsupervised anomaly detection tasks in the DevOps field. We prove this by comparison with other models on real-world datasets collected from production.
- We enhance the Temporal Fusion Transformer model by integrating input sequences in a probabilistic way, which improves performance.
- We propose a framework called **TFTOps** that is suitable for near real-time, concurrent anomaly detection of hundreds of different series.

II. RELATED WORK

In this section, we will discuss the development of anomaly detection methods and the usage of these methods in DevOps/AIOps domain.

A. ANOMALY DETECTION

Anomaly detection is an active field of research in which many applications and solutions being proposed every year in the past decades.

A natural solution to the time series anomaly detection problem is to utilize the "normal" time series to build a model capable of predicting the following items when the system is running normally. Given such a model and an observation of time series, one can use the observation as the model's input, and get the prediction indicating what the following values could be under the system's "normal" situation. If the observed values of future time series violate the predicted ones, the operator should mark the timestamp as "abnormal". Under this setup, two essential problems need resolving.

- Finding a good model that can correctly predict our target sequence. Depending on the characteristics of different targets, "good" models often vary in scales, costs, and even methodologies.
- Finding a method to examine whether a violation case is "abnormal", or just a false positive/outlier. Such methods mainly refer to various thresholding techniques, including static and dynamic (or self-adaptive) thresholding.

Traditional time series analysis methods often tend to use simple statistic methods to model time series. Early in the 1990s, the ARIMA model [4] was proposed. The full name of ARIMA is "AutoRegressive Integrated Moving Average", which uses a differencing operation (or "integrate") to eliminate the non-stationarity if possible. To deal with seasonal components, a seasonal-differencing technique is also introduced. Given input, an ARIMA model generates a scalar value, and operators often compare the prediction with a static threshold to determine whether the prediction-observation pair is abnormal.

The naïve ARIMA method experiences difficulties when dealing with complex time series. However, the series can be non-stationary even after the integration process. Running a second-order differencing $y_t* = y_t - 2y_{t-1} + y_{t+2}$ helps to eliminate other components, but the complexity of an ARIMA model is still challenged by many real-world datasets.

On the other hand, some efforts were made to decompose the time series into different interpretable components. Generally speaking, one can write a time series as a composition of 4 series: a level component, a trend component (T), a seasonal component (S), and an error term (E), which is the gap between the model and actual observations. Different models make different underlying mathematical assumptions about the methods to compute and combine those components, forming a family that is called ETS models. Hyndman et. al. [5] provides a detailed view of the ETS family.

However, these traditional models suffer from performance degradation when the target becomes complex, or when we are unable to provide accurate prior knowledge. For example, consider a time series describing the number of customers in a resort. One can easily conclude that it should be busy on weekends, and have fewer customers during weekdays, thus the seasonal component in ARIMA/ETS should have a period of 7 days. However, this assumption breaks when Christmas is coming. The input of traditional models only includes the target series itself and several hyper-parameters. Their lack of conditional inputs makes the models harder to generalize to different circumstances.

Since the 2010s, there has been a growing interest in neural network-based methods for various machine learning applications, including time series prediction and anomaly detection. As the previous example suggests, time series often come with prior knowledge, while neural networks are well-suited to incorporate this information into the models. Thanks to its nature, one can skip the intensive work of choosing feasible features from all sorts of prior knowledge, and let the stacked layers in neural networks do the job automatically.

During past decades, researchers have proposed a variety of neural network models, and it is hard to thoroughly introduce every different directions. To make a comparison with our proposed TFTOps, we mainly focus on the works that are close to our model in target, approach, and scalability potential. In other words, we introduce some models that have the following properties:

1) Uses Deep neural network (DNN) to perform time sequence prediction. This ensures that the model is somehow generalizable enough to capture fuzzy and versatile sequences.

2) Integrates static information and other "known" variables with input. These information, acting as prior knowledge, will contribute to the model's overall performance.
3) Directly predict the future value of the sequence (unsupervised), other than only predicting a manually attached "anomaly" label. This ensures the availability of model when exact anomaly labels can hardly be retrieved.

Condition 1) and 3) indicate a task called multi-horizon sequence forecasting. Given a time sequence as input, the DNN model should generate its prediction for the possible values ahead.

In the field of DNN, there are two main approaches to solve the multi-horizon forecasting task. The first approach in this direction features the autoregressive models such as Seq2seq [6] and DeepAR [7]. Autoregressive models usually base on Long-short Term Memory (LSTM) cells [8] and their variations, such as GRU [9]. Deep AR [7] uses stacked LSTM layers to generate parameters (mean and variation) of a Gaussian distribution, then samples from that distribution to determine the one-step-ahead output. Deep State-Space models [10] implements a similar approach, while modeling the Gaussian distribution parameters with a Variational Autoencoder(VAE) [11]. More recent researches focus on Transformers, for example [12] proposed the use of transformers in time series tasks and introduced convolutional layers to reduce memory footprints. Fan et.al. [13] proposed a multi-modal attention mechanism to enhance LSTM encoders, which provides better context to a bi-LSTM decoder. These models are built to solve the "one-step-ahead" prediction problem: accept the previous sequence as input from $0..t$, then predict the value of $t + 1$. After that, one should send the prediction at $t + 1$ back to the model to get the next timestep after $t + 1$, which is $t + 2$. This process is repeated, until all timesteps $t + 1, \ldots, t + n$ are iteratively generated. While straightforward, the efficiency of these models can be the bottleneck in certain situations. Inevitably, we must call the model n times to get n predictions, which results in an $O(n)$ complexity. Caching the decoder part can mitigate the problem.

On the contrary, non-autoregressive models generate a sequence of forecasts for a fixed number of horizons concurrently. A typical model also resembles the normal approach of using an encoder to generate a hidden representation for the whole input. Common choices of encoders include feed-forward CNN,Autoencoder [14], LSTM, transformer, or a combination of these models [15]. Upon generating the representation vector, a decoder will process it to acquire future predictions. The whole model is end-to-end trainable. For example, the Multi-horizon Quantile Recurrent Forecaster (MQRNN) [16] proposed two different encoder structures (CNN and LSTM), and generated output based on the encoded sequence via an MLP for each horizon. Temporal Fusion Trasnformers [3], unlike [12], directly predicts the $t + 1 \ldots t + n$ timesteps using a transformer-based model in a single pass. Note that both models (MQRNN and TFT) generate the whole group of multi-horizon output at the same time, avoiding the self-regression scheme. Besides, both models chose "known" variables as extra input of their decoders, which satisfies 2).

In terms of model structure, our TFTOps mainly inherits ideas from TFT model [3] and [17]'s probabilistic input.

B. ANOMALY DETECTION IN AIOPS

DevOps relies on different aspects to monitor the status of system. Main datasources include Application logs [18], distributed traces [19] and KPI metrics. More comprehensive results can be found in [20]. In this paper, we mainly focus on researches about KPI metrics (time series).

Currently, the most widely accepted toolchains include Prometheus [21] and Grafana [22] which have user-friendly interfaces and supported by a powerful query language (PromQL [23]). However, those platforms currently only support naïve models as built-in functions, such as linear regression and statistical tests. These approaches are suitable for most periodic data, while failing to solve complex cases where an assumption of underlying data distribution is unavailable.

In the past decades, machine learning approaches are intensively investigated for metric prediction. Supervised methods include decision trees [24], [25], [26] or Bayesian classifiers [27], [28], [29]. While achieving high accuracy in some cases, the performance of supervised methods heavily depends on accurately labeled data. Unsupervised methods include LOF(local outlier factor) [30], clustering [31] and PCA [32] are developed for the cases where labeled data is unavailable.

Being successful in other fields, such as computer graphics and machine translation, neural network methods are gaining notice in the AIOps context since 2015. For example, Monni et. al. [33] built an anomaly detector based on restricted Boltzmann machine [34]; Lin et. al. using a Learning-to-Rank model [35] to combine the result of LSTM and random forest. In more recent researches, attention mechanism also took place in predicting long-term time series [36], [37]. Except directly predicting the target, operators can also use neural networks to generate reliable fake sequences [38] to augment the dataset. However, the performance of neural networks is often a concern to the operators.

Generally speaking, previous researchers mainly focused on predicting the trend of KPIs(Key Performance Indicators). For example, Microsoft's Spectral Residual CNN KPI anomaly detection model [39] treats each KPI time sequence separately: an independent model is trained on each KPI sequence. Since KPI sequences can be defined quite differently, using different model parameters to predict them is a natural choice. Also, since the number of such KPI sequences is few, one can easily afford the cost of training and hosting several models. However, in typical microservice settings, there are hundreds of pods and services hosted for different purposes. When the operators try to make fine-grained predictions, they will likely meet a dilemma: using a neural network to accurately track a pod/service's metric, such as disk and memory usage, often consumes more resources than the pod/service itself. This dilemma prevents neural network models from participating in fine-grained prediction tasks.

Recently, transformer-based models have shown superior performance in neural machine translation and other text-related fields. Its power in processing sequences also attracts the attention of researchers in time-series fields. Since 2020, transformer-based models have formed a line of research on long-term time series forecasting. For example, Informer [40] proposed an efficient ProbSparse self-attention mechanism, which solved the performance cap of long time series(output length > 50); Autoformer [41] proposed a decomposition architecture and auto-correlation mechanism to further aggregate sub-series level representations.

However, in DevOps settings, it is doubtful that long-term time series forecasting is necessary. As proposed by [40], the prediction error quickly rises as the output sequence length increases. In production environments, DevOps tends to shrink the length directly instead. Discovering abnormal circumstances hours ahead is good enough; in this case, a shorter time series(output length < 50) would suffice. Despite using similar transformer and self-attention mechanisms, this paper mainly explores another dimension: the "width" of transformer models in the practical DevOps field. In other words, this refers to the ability to compute a group of similar metrics using the same model structure and parameters.

III. METHODOLOGY

This section describes the architecture and training scheme of our proposed model TFTOps.

A. NETWORK ARCHITECTURE

The model consists of three main parts: The input embedding layer, the Input LSTM layer, and the temporal self-attention layer. The architecture is shown in Fig.1, and each components are connected according to Algorithm 1. In the following sections, we will also present the structure of each component in a bottom-up order. Also, we will explain the rationales behind our choices.

B. MULTI-INPUT AND PREPROCESSING

We begin with basic attributes that defines a model: input and expected output.

In a microservice system, the metrics time series are often strongly entangled together. For example, for a service S, a growing network traffic indicates its load is increasing and will naturally lead to a peak in CPU usage and disk I/O. Depending on the system architecture, such inter-relationships often tend to have some delay, making it even more valuable in predicting future metrics.

As in [3], we divide the input into two parts: "static input" and "variable input". The general rules of division are:

```
Algorithm 1 Forward Pass of TFTOps Model
Require: Input variables $v_{obs}, v_{known}, v_{static}$

  $\xi^{(i)} \leftarrow InputGRN(v^{(i)})$: Feed variable inputs $v_{obs}$, $v_{known}$ into GRN units to get encoded feature for variable $i$.
  $\tilde{\xi} \leftarrow \text{VSelect}(\xi, v_s static)$: Encoded features $\xi$ enter a variable selection network VSelect, where we combine them with static inputs $v_{static}$, and acquire the merged feature.
  $\tilde{\phi} \leftarrow LSTM(\tilde{\xi}, v_{static})$, $\Phi \leftarrow GRN(\tilde{\phi}, \tilde{\xi})$: An LSTM layer plus a residual layer for extracting low-level information.
  $\psi \leftarrow \text{Attn}(\Phi)$, $\tilde{\psi} \leftarrow \text{LayerNorm}(\tilde{\phi} + GLU(\psi))$: A masked self-attention layer for modeling high-level dependencies. Another residual connection to restore low-level info.
  $\hat{y} \leftarrow \text{Dense}(\tilde{\psi})$: The final dense output layer.
```

TABLE 1. An example of categorical variables.

name	value1	value2	value3	...
business	k8s	monitor	aiops	...
fstype	ext4	ntfs	...	...
instance	10.17.xx.1	10.17.xx.2	10.17.xx.254	...
mountpoint	/local	/(root)	/mnt	...

- Static inputs
  In Prometheus, each time series has some associated attributes ("labels"). Labels are read-only categorical variables describing the features of a given time series created upon the initialization of that series. When extracting metrics from a server/pod, the node exporter automatically attaches labels to the metrics; maintainers can also customize the labels. For example, the following time series from Prometheus' node exporter has 4 pre-defined static inputs: business, fstype, instance(IP), and mountpoint.

```
    node_filesystem_avail_bytes{
        business=``k8s'',
        fstype=``ext4'',
        instance=``10.17.xx.xx'',
        mountpoint=``/local''
        }
```

  These inputs are valuable in predicting future disk usage. The "Business" label is manually assigned and represents the department to which the node belongs. Therefore, we collect all of the time series under the same metric from a microservice system and define 4 categorical variables according to the unique values in the node exporter. After choosing those categorical variables, we transform them into one-hot features while maintaining a mapping for all the unique values we saw in this process. See 1 for an example.
- Variable inputs
  Variable inputs are the main components that build up a time series. In our monitoring system, there are 2 types of input that will change over time: values of

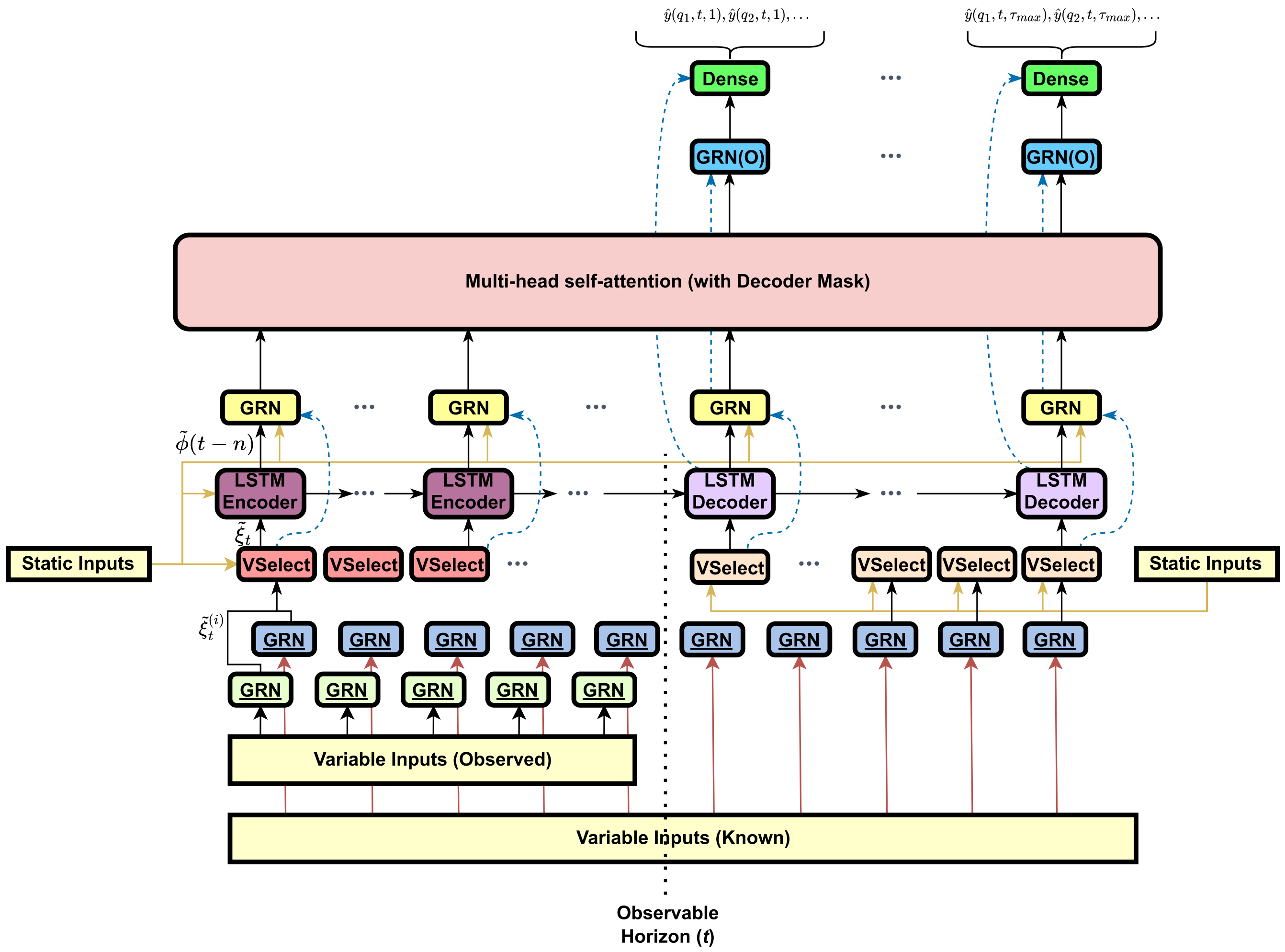

$\hat{y}(q_1,t,1),\hat{y}(q_2,t,1),\ldots$
$\hat{y}(q_1,t,\tau_{max}),\hat{y}(q_2,t,\tau_{max}),\ldots$
Dense
GRN(O)
Multi-head self-attention (with Decoder Mask)
GRN
$\tilde{\phi}(t-n)$
LSTM Encoder
LSTM Decoder
$\tilde{\xi}_t$
$\tilde{\xi}_t^{(i)}$
Static Inputs
VSelect
Variable Inputs (Observed)
Variable Inputs (Known)
Observable Horizon (t)

FIGURE 1. **The architecture of our TFTOps model. Blue dashed lines denote skip connections. Cells with the same color shares the same weight across different timesteps.**

different metrics and the timestamp that produces these values. Many other variables, such as hour, day of the week, and whether it is a holiday, can be generated from timestamps. We denote the metrics as "observed variables" and timestamp-related inputs as "known variables".

The main difference between those 2 types of inputs is whether we can "foresee" the exact future value. Since Prometheus periodically fetches metrics from sources, we have a fixed timestamp-delta between neighboring observations. We can conduct the value precisely and generate other fields, such as the hour/day of the week, for any future timesteps. On the other hand, the future value of metrics remains unknown, as it's our model's job to predict them.

Therefore, unlike traditional RNN-based models, the input dimensionality of the transformer encoder and decoder in TFTOps are NOT the same. We resolved this by the same variable selection network in [3], which conceptually serves as a trainable weighted average among input features.

As shown in figure 1, the variable features (yellow) are fed into two Gated Residual Networks(GRN). As in [3], the GRN block is a basic building block of our model. In this process, two types of GRNs are used: one(green) processes the observed variable inputs(input range is $[1..t]$), and another GRN(blue) processes the known variable inputs(input range is $[1..t+\tau]$).

C. GATED RESIDUAL NETWORKS

The idea behind GRN is like a residual network: the model chooses to apply a non-linear process when needed. As in 2, a GRN unit receives two vectors as its input: a primary input a, and an optional context c. We compute the layer's output as follows:

$$\begin{aligned}\mathrm{GRN}(a,c) &= \mathrm{LayerNorm}(a+\mathrm{GLU}(\eta_1))\\ \eta_1 &= W_1\eta_2+b_1\\ \eta_2 &= ELU(W_2a+W_3c+b_2),\end{aligned}$$

where GLU stands for Gated Linear Units [42], and ELU stands for Exponential Linear Unit. If we do not provide the context, then context c is replaced with a zero vector $c = \mathbf{0}$

$$GLU(x) = \sigma(W_{g1}x+b_{g1})\odot(W_{g2}x+b_{g2})$$

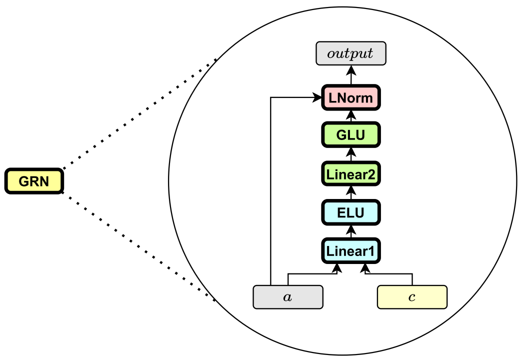


FIGURE 2. The detailed architecture of a GRN Unit. Each GRN unit contains 2 "linear+activation" structure, a LayerNorm, and a skip connection. Contextual input (*c*) is optional, and does not present in the skip connection.

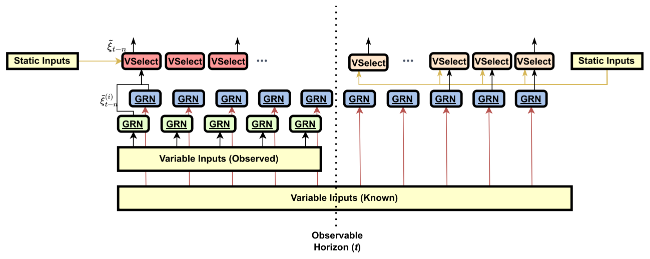


FIGURE 3. The architecture of input embedding layer. GRN layers with the same color share the same group of parameters.

$\odot$ means element-wise Hadamard product. Through a GLU layer, the model can suppress any useless feature in x, which means doing variable selection in a differentiable way.

The ELU activation is conceptually similar with the ReLU activation, but is smooth and differentiable:

$$\mathrm{ELU}(x) = \begin{cases} x & x > 0 \\ \alpha(\exp(x) - 1) & x \leq 0 \end{cases}$$

D. INPUT EMBEDDING LAYER

This layer is the first part of tft model, and it directly process the input features of various types and shapes. It consists of embedding layers and a variable selection network. 3

1) EMBEDDING LAYERS

Each different input variable $i \in \{1, 2, \ldots, m\}$ uses its own embedding layer. The input variable, no matter it is one-hot or real-valued, is linearly transformed into a fixed-dimension vector $\boldsymbol{\xi}_t^{(i)} \in \mathbb{R}^{d_{model}}$:

$$\boldsymbol{\xi}_t^{(i)} = W^{(i)} X_t^{(i)}$$

An additional layer of GRN introduces a non-linear process, transforming $\boldsymbol{\xi}_t^{(i)}$ into $\tilde{\boldsymbol{\xi}}^{(i)}$:

$$\tilde{\boldsymbol{\xi}}_t^{(i)} = \mathrm{GRN}_{\boldsymbol{\xi}_t^{(i)}} \boldsymbol{\xi}_t^{(i)}$$

For each variable i, the weights of linear transform $W^{(i)}$ and $\mathrm{GRN}_{\boldsymbol{\xi}^{(i)}}$ are shared across all timesteps t.

2) VARIABLE SELECTION NETWORK

After the embedding layer, the transformed $\tilde{\boldsymbol{\xi}}_t^{(i)}$ s are fed into a variable selection network. The variable selection network flattens the output of different variables from embedding layer, and builts its own input Ξ_t:

$$\Xi_t = [\boldsymbol{\xi}_t^{(1)^T}, \boldsymbol{\xi}_t^{(2)^T}, \ldots, \boldsymbol{\xi}_t^{(m_\chi)^T}]$$

Also in this layer, a context vector $\boldsymbol{c}_s$ from static inputs is introduced to build a weight vector:

$$\boldsymbol{v}_{\chi t} = \mathrm{Softmax}(\mathrm{GRN}(\Xi_t, \boldsymbol{c}_s))$$

We combine the weight $\boldsymbol{v}_{\chi t}$ with the GRN-transformed feature $\tilde{\boldsymbol{\xi}}_t^{(i)}$ to get the final output at timestep t:

$$\tilde{\boldsymbol{\xi}}_t = \sum_{j=1}^{m_\chi} \boldsymbol{v}_{\chi t}^{(i)} \tilde{\boldsymbol{\xi}}_t^{(i)}, \; \tilde{\boldsymbol{\xi}}_t \in \mathbb{R}^{d_{model}}$$

After that, each timestep, regardless of how many input features it has, is transformed into a vector with fixed dimension $\mathbb{R}^{d_{model}}$, and is ready for further computations.

E. PROBABILISTIC INPUT

The observation frequency in AIOps is much higher than in traditional time-series prediction tasks (such as grocery sales [43] and climate change). Most monitoring systems collect metric values on a minute-level basis; thus, a typical period (a day) would contain $24 * 60 = 1440$ data points. Neither sequential models (RNNs) nor transformer models can easily handle such a long input series. A workaround is to resample the whole series with a larger interval; however, this resampling means ignoring (or averaging) all the observed values in between, which is not advantageous.

To resolve this problem, we leverage the "probabilistic input" idea from [17] to enhance the previous input schema. We illustrate this probabilistic pre-processing method by a real-world task below.

One of our clients has a Prometheus monitoring system whose retrieving interval (the time difference between two neighboring data points) is $\Delta t = 60s$. They wish to predict the disk usage after 4 hours, which would introduce a gigantic decoder ($l_{decoder} = 240$) and an encoder even larger than $l_{decoder}$. To shrink the size of the encoder, we performed the following pre-processing steps to implement the "probabilistic input" scheme:

1) Firstly, we manually determine an interval I according to domain knowledge. For example, in this task, we choose $I = 20$, which is combining 20 inputs into ONE distributional input.
2) Secondly, we normalize the real-valued inputs. After normalization, the model split the values into $n_{\text{bins}} = 10$ bins. n_{bins} is a hyperparameter determined through grid search, and its typical search range is $[0.05, 0.3] * I$, where I is the previously mentioned aggregation interval. Let $b(t) : \mathbb{R} \to \mathbb{R}^{n_{\text{bins}}}$ be the binarization

function which maps the real-valued inputs $x(t)$ to one-hot input $b(t)$.

3) Cut the input sequence according to the previously mentioned interval I. Each timestamp t now corresponds to a time window that starts from $t-I+1$ and ends with t itself. We transform the values in a window into a one-hot vector, then take an average to build a probabilistic input series:

$$P_x(t) = \frac{1}{I}[b(t-I+1) + b(t-I+2) + \ldots + b(t)]$$

Similarly, for preceding inputs $P_x(t+k)$, we have:

$$P_x(t+k) = \frac{1}{I}[b(t+k(I-1)+1) + \ldots + b(t+kI)]$$

After the pre-processing, each time window (R^I)in our input series is transformed into a single vector $R^{1\times n_{\text{bins}}}$. Concatenating those new vectors, we get a new time series P_x. This new time series P_x has a period length of $I \cdot \Delta t$; each element in this new time series can be viewed as a sample from a multinoulli distribution. Note that we only transform the encoder inputs; the output sequence of the decoder is sub-sampled by I as usual. In order to predict the value of our time series after 4 hours, our new decoder only needs to process 12 timesteps after the sub-sampling, while each timestep has more features. As our input embedding layer suggests, adding features to our current input only affects the dimensionality of Ws and the variable selection network. Therefore, the addition of probabilistic inputs will only have a negligible impact on the efficiency of the model compared to a model sampled with interval I. Compared to the original model without interval sampling, the overall efficiency of the model is greatly improved as the length of the encoder is reduced by $20\times$.

We apply a linear layer to transform the $\mathbb{R}^{n_{\text{bins}}}$ vector P_j into a $\mathbb{R}^{d_{model}}$ vector $\boldsymbol{\xi}_t^{(P_j)}$. Then, this new feature is fed into its own GRN like other non-probabilistic features:

$$\tilde{\boldsymbol{\xi}}_t^{(P_j)} = \text{GRN}_{\tilde{\xi}^{(P_j)}}(\boldsymbol{\xi}_t^{(P_j)})$$

The processed $\tilde{\boldsymbol{\xi}}_t^{(P_j)}$ is then feed into the variable selection layer **along with** the original feature $\tilde{\boldsymbol{\xi}}_t^{(j)}$. We also keep the original feature alongside the probabilistic feature for capturing the exact value, which is beneficial especially when the feature itself is a target for prediction.

Note that we only use the "probabilistic" pre-processing scheme when dealing with observed variables. Known variables, which often tightly related with timestamps, are resampled with a larger interval as usual.

F. INPUT LSTM

As in figure 1, the outputs of the input embedding layer at every timestep ($\tilde{\boldsymbol{\xi}}$) are fed into an LSTM encoder/decoder. This LSTM network [44] helps the model to capture the hidden informations in consecutive values.

The LSTM output at each timestep (encoder and decoder) is denoted as $\phi(t)$. As in [3], a residual layer is used for robustness, which we mark as blue dashed lines. The residual layer directly sends $\tilde{\boldsymbol{\xi}}$ into the LSTM output. On the other hand, if our model find the LSTM encoder/decoder is not beneficial, the Gated Linear Unit(GLU) activation would suppress its impact:

$$\tilde{\boldsymbol{\phi}}(t,n) = \text{LayerNorm}\left(\tilde{\boldsymbol{\xi}}_{t+n} + \text{GLU}(\boldsymbol{\phi}(t,n))\right)$$

The produced output $\tilde{\boldsymbol{\phi}}(t,n)$ is merged with the static information via a GRN layer. As in input embedding layer, the static information serves as context in GRN:

$$\boldsymbol{\theta}(t,n) = \text{GRN}(\tilde{\boldsymbol{\phi}}(t,n), \boldsymbol{c}_e)$$

Through input embedding and LSTM, we extract the local information and put them into $\boldsymbol{\theta}(t,n)$

We concatenate $\boldsymbol{\theta}(t,n)$ from different timesteps to build the input of Temporal self-attention layer $\boldsymbol{\Theta}(t)$:

$$\boldsymbol{\Theta}(t) = [\boldsymbol{\theta}(t,-k)^T, \ldots, \boldsymbol{\theta}(t,\tau)]^T$$

After this layer, informations in the series short-term relationships are extracted into the output features.

G. MULTI-HEAD ATTENTION

As in TFT [3], TFTOps implements a self-attention mechanism. This mechanism, which is inherited from the original Transformer model [45], helps the model learn the long-term relationships among input timesteps.

Generally speaking, the attention mechanism is a scaler upon "values" $\boldsymbol{V} \in \mathbb{R}^{N\times dv}$. The scaling factor is determined by relationships between "keys" $\boldsymbol{K} \in \mathbb{R}^{N\times d_{attn}}$ and "queries" $\boldsymbol{Q} \in \mathbb{R}^{N\times d_{attn}}$:

$$\text{Attention}(Q,K,V) = A(Q,K)V$$

where $A()$ is the "attention function". The most common choice for $A()$ is the scaled dot-product function:

$$A(\boldsymbol{Q},\boldsymbol{K}) = \text{Softmax}(\boldsymbol{QK})^T/\sqrt{d_{attn}}$$

Multi-head attention layers are built by replicating the dot-product attention several times. A single dot-product attention is called a "head", and the number of heads is denoted by a hyperparameter m_H. All heads share the same input tuple($\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$), but rescale it by different, head-specific weight $\boldsymbol{W}_Q^{(h)}, \boldsymbol{W}_K^{(h)}, \boldsymbol{W}_V^{(h)}$ before calling the attention mechanism:

$$\boldsymbol{H}_h = \text{Attention}(\boldsymbol{QW}_Q^{(h)}, \boldsymbol{KW}_K^{(h)}, \boldsymbol{VW}_V^{(h)})$$

After done computing all $\boldsymbol{H}_h$, the values are linearly combined through another weight matrix $\boldsymbol{W}_H$ to build the final output of multi-head attention:

$$\text{InterpretableMultiHead}(Q,K,V) = [\boldsymbol{H}_1, \ldots, \boldsymbol{H}_{m_H}]\boldsymbol{W}_H.$$

We made the same adjustments against traditional multi-head attention layers, as in [3]. The core idea is to "force" all heads to use the same value of $\boldsymbol{V}$ and $\boldsymbol{W}_V$. The weight matrix for keys $\boldsymbol{W}_K$ and queries $\boldsymbol{W}_Q$ remain different among heads,

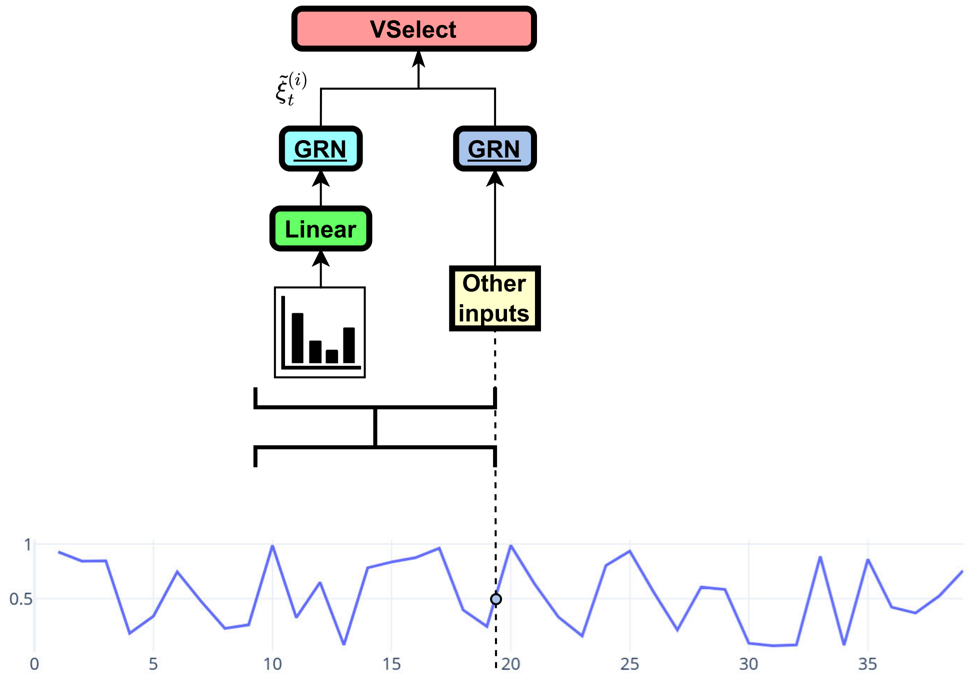


FIGURE 4. The architecture of our probabilistic input scheme. The probabilistic feature is extracted across a sub-sampled time window. The dashed vertical line marks the end of a sub-sampled window, where the other non-probabilistic variables (including "known" and "observed" variables) are extracted. After processed by its own GRU layer, each variable equally enters the variable selection layer.

and we average their scales **before** multiplying them with $\boldsymbol{VW}_V$:

$$\begin{aligned}\tilde{H} &= \tilde{A}(\boldsymbol{Q},\boldsymbol{K})\boldsymbol{VW}_V \\ &= \left\{\frac{1}{H}\sum_{h=1}^{m_H} \mathrm{A}\left(\boldsymbol{QW}_Q^{(h)},\boldsymbol{KW}_K^{(h)}\right)\right\}\boldsymbol{VW}_V) \\ &= \frac{1}{H}\sum_{h=1}^{m_H}\mathrm{Attention}\left(\boldsymbol{QW}_Q^{(h)},\boldsymbol{KW}_K^{(h)},\boldsymbol{VW}_V\right)\end{aligned}$$

Thus, each head attends to the same input features corresponding $\boldsymbol{V}$ while learning different patterns through their own $\boldsymbol{W}_Q^{(h)}$ and $\boldsymbol{W}_K^{(h)}$. This approach is similar to convolutional neural networks(CNNs). Here, attention heads act like convolutional kernels in CNN, and the number of heads m_H is comparable with the number of "channels" in convolution layers. By this way, the total number of trainable parameters and computation overhead are both decreased.

H. MULTI-HEAD SELF-ATTENTION LAYER

By feeding the output of GRNs into aforementioned interpretable self-attention layer, we model the long-term dependencies with a stacked multi-head self-attention.

We apply interpretable multi-head self-attention only once. All previous GRN results, range from the first timestep in input sequence to the last timestep of prediction ($t \in [0..t+\tau]$), are fed into the self-attention layer $\boldsymbol{\Theta}$. The same $\boldsymbol{\Theta}(t)$ is fed into all three slots: query, key and value. Note that, to prevent leaking of training data, the decoder timesteps in self-attentions are masked [12], [45]. For example, at timestep $t+\tau-1$, the model sees only the input from $t \in [0..t+\tau-2]$, but not $t+\tau$. This ensures a causal information flow.

$$\boldsymbol{B}(t) = \mathrm{InterpretableMultiHead}(\boldsymbol{\Theta}(t),\boldsymbol{\Theta}(t),\boldsymbol{\Theta}(t))$$

$\boldsymbol{B}(t)$ has the same dimensionality as $\boldsymbol{\Theta}(t)$. We decompose it into $\boldsymbol{B}(t) = [\boldsymbol{\beta}(t,-k)^T,\ldots,\boldsymbol{\beta}(t,\tau)]^T$

This self-attention layer can also be stacked, as mentioned in [45]. When stacking, the following layer directly deals with the output of previous self-attention layer; this time, we do not include residual connection and/or static contexts.

Again, in order to preserve the local information learned by RNN, we use a residual connection. This is marked with blue dashed line in 1. The transformer output and the residual are concatenated channel-wise, then fed into another GRN block(blue):

$$\boldsymbol{\psi}(t,n) = \mathrm{GRN}\left(\mathrm{LayerNorm}\left(\tilde{\boldsymbol{\theta}}_{t+n} + \mathrm{GLU}(\boldsymbol{\beta}(t,n))\right)\right)$$

I. OUTPUT LAYER AND QUANTILE LOSS

We provide another skip connection which skips the entire transformer block, directly connect the output of LSTM to the dense layer.

$$\tilde{\boldsymbol{\psi}}(t,n) = \mathrm{LayerNorm}\left(\tilde{\boldsymbol{\phi}}_{t,n} + \mathrm{GLU}(\boldsymbol{\psi}(t,n))\right)$$

This new embedding $\tilde{\boldsymbol{\psi}}(t,n)$ passes through a linear layer to produce the final output prediction at this timestep. Given current timestep t and the future timesteps we want to predict

$0 < \tau < \tau_{max}$, we have:

$$\hat{y}(t,\tau) = \boldsymbol{W}\tilde{\boldsymbol{\psi}}(t,n) + b$$

The shape of prediction at timestep τ ($\hat{y}(t,\tau)$) is $\mathbb{R}^{n_{\text{target}}}$, where n_{target} denotes the number of features that we want to predict.

To better fit the requirements of anomaly detection, we choose the averaged quantile loss. We define a sequence of n_q quantiles between (0, 1): $0 < q_1 < q_2 < \ldots < q_{n_q} < 1$. The output dense layer is enlarged channel-wise to get outputs at each quantile, which makes the output shape become $\hat{y} \in \mathbb{R}^{n_{\text{target}} \times n_q}$. For each quantile q_i, its output $\hat{y}_q$ is the i-th dimension of $\hat{y}$. We define the output value and its corresponding "quantile loss" function $QL(y, \hat{y}_q, q)$ as:

$$\hat{y}_q(t,\tau) = \boldsymbol{W}_q\tilde{\boldsymbol{\psi}}(t,n) + b_q \tag{1}$$

$$QL(y,\hat{y},q) = q(y-\hat{y}_q)_+ + (1-q)(\hat{y}_q - y)_+ \tag{2}$$

The quantile q is customizable according to the requirements of different jobs and domain knowledge. For example, a common choice is 0.1, 0.5, 0.9, which means we have a pessimistic estimation ($q = 0.1$), a balanced estimation ($q = 0.5$), and an optimistic estimation (q=0.9). The pessimistic and optimistic predictions work as lower/upper bounds of the observed value. On the other hand, when given the quantile 0.5, the model acts as a standard regression model with L_1 loss.

Our final loss is the average loss across all quantiles $q_1, \ldots, q_n$.

J. OUTPUT SCHEMES AND DETECTION MODES

There are two primary use cases of TFTOps.

Firstly, we can automatically get an adaptive threshold for a certain metric with the quantile loss function. As the quantile loss function suggests, quantile $q = 0.5$ is the output of a standard regression model; at higher quantiles ($0.5 < q < 1$), the model tends to output a higher prediction to avoid being heavily punished by the loss function. Likewise, lower quantiles ($0 < q < 0.5$) would lead to a lower prediction. Assuming our model can accurately predict the metric value in "normal" cases (quantile $q = 0.5$), then any value higher than the highest quantile ($q = 0.9$) or lower than the lowest quantile ($q = 0.1$) can be considered "abnormal". We can adjust the quantiles to balance precision and recall. Generally speaking, squeezing the upper and lower quantiles towards 0.5 improves recall but harms precision, for it would introduce some false positives; stretch them towards 0 or 1 will improve precision instead. For robustness, the "abnormal state" alert should be triggered when the prediction violates the rules above for a consecutive certain period. Typically, we choose the period as 5∗sampling_interval, which means 5× consecutive violations will trigger an alarm in Prometheus.

Secondly, the TFTOps model itself can also serve as a prediction model. Traditionally, such prediction is retrieved by extracting trends via simple regression models (such as linear regression) from the last hours, then extending the extracted trend. However, this naïve method does not consider external messages such as date or time, thus introducing a high chance of false alarms. In our TFTOps model, the loss function for the "normal" quantile ($q = 0.5$) is the same as the traditional L_1 loss. The prediction at this quantile is also suitable for directly forecasting the metric in the near future. With a user-defined threshold, the TFTOps model can predict abnormal cases (for example, CPU use rate > 80%) several minutes before the anomaly happens.

IV. EXPERIMENTS

A. EXPERIMENT SETUP

We implemented TFTOps under tensorflow framework. Both training and prediction are performed inside the same kubernetes pod, which is hosted on a node with Intel(R) Xeon(R) Gold 6278C and no GPUs.

Our dataset is retrieved from a Prometheus platform which monitors a working kubernetes cluster with hundreds of nodes and thousands of running pods. Because prometheus uses an in-memory TSDB database, which is a temporary storage and does not guarantee persistence, our training set is periodically retrieved from a PostgreSQL database, which serves as a persistent data storage of Prometheus.

B. DATASET OVERVIEW

1) PROMETHEUS NODE EXPORTER (PNE)

The Prometheus monitoring platform scrapes metrics from *node_exporter*s in kubernetes system every 20 seconds.

Our first experiment chose "*node_filesystem_free_bytes*" metric as the target. The metric indicates the amount of free space in each mounted device on k8s nodes. This metric is tightly connected with system load, thus receives great attention from our operators. This metric would certainly go up as system load increases; however, there are many reasons to cause the increase of metric, and their severity differs greatly. For example, when the system load is reaching a local peak, an increase in *node_filesystem_free_bytes* is perfectly normal; when the system is experiencing a DDoS attack, or some bug occurs in load-balancing components, an increase in the same metric would eventually break the system. Discriminating the two different cases is proved to be a challenge, and we tried to resolve it by TFTOps.

We can easily find other useful metrics in *node_exporter*, such as *node_network_receive_bytes_total* which could be useful in predicting future values. We integrated those metrics in a group of experiment to clarify whether providing extra time series are helpful in this job.

Generally speaking, the sequences are very likely to be monotonic in given time window, which makes the traditional methods (ARIMA, ETS) suitable for this task. However, traditional approaches require fitting a model per prediction task, greatly slowing down the prediction process. On contrary, inference through a neural network is more efficient.

TABLE 2. Features of PNE dataset.

variable	type	description	example
business	static	the department which owns the instance	usercenter
device	static	the device name of disk	/dev/vda2
fstype	static	type of filesystem	ext4
instance	static	ip of the server	10.13.x.x
location	static	data center of the server	DC02
target	observed	The remaining space (GB)	16.43
cpu	observed	CPU usage over the last minute	0.031
network	observed	average network traffic (kB) per second, over the last minute	1028.4
weekday	known	the weekday, can be derived from timestamp	6
hour	known	the hour of the day	14
holiday	known	Boolean, whether the day is a holiday	0

The input variables are extracted directly from Prometheus sequence, which contains a set of labels to describe the sequence's properties. We used PromQL to extract the whole sequence as a list of data points extract at different time from different servers. Each point contains timestamp, value(bytes), and a set of descriptive variables indicating the situation of the source. We use the following variables in our experiment:

Note that, since the same timestamp will definitely not appear in both train and test datatsets, we do NOT use the timestamp itself as a feature.

2) REDIS CONNECT(RDC)

The second experiment aims to predict the number of active connections per redis job in a redis cluster. In out production Redis cluster, a crontab job is running to retrieve the number of connections once per minute, and send the message to a kafka topic for further analysis. Originally, the operators use a nåive alert system based on thresholding, which is proved to be annoying, since the alert is triggered by false positives from time to time. We directly consume the topic, and record the historical values in a database. This metric is somehow proportional to the system's KPI because it is closely related system load and user traffic. Traditional temporal-filtering based models failed to deal with such KPI-related metrics.

Due to the nature of redis connections, most of sequences in this dataset are oscillatory. Traditional methods fails to converge (ARIMA), or performs poorly (ETS) when processing such sequences.

We use the following variables in our experiment:

TABLE 3. Features of RDC dataset.

variable	type	description	example
id	static	the redis job id	11250
target	observed	the number of connections of certain job	694
weekday	known	the weekday derived from timestamp	3
hour	known	the hour in day	18
is_holiday	known	whether the day is a holiday	1

3) BENCHMARKS

We compare TFTOps to different types of models for multi-horizon forecasting. For models which base on neural networks, we used roughly the same number of search iterations to conduct hyperparameter optimization over a pre-defined search space centered around their default parameters (mentioned by their lrespective paper).

A brief introduction of relevant models and their source code (if available):

- **ARIMA [4]**: ARIMA is a traditional statistical model for time series analysis. Note that there is no guarantee that all input sequences should be stationary; for the non-stationary sequences which cause the ARIMA models to fail, we use the last observed value as a substitution.
- **ETS**: Another traditional statistical model, which decomposes time series into error, trend and seasonal components. Both ARIMA and ETS does NOT require additional features.
- **MQRNN [16]**: A recurrent network for multi-horizon forecasting. Its encoder is the same as seq2seq LSTM encoder (but without attention) and uses the same hyperparameter search space.
- **RoughAE [14]**: A neural network model based on the denoising autoencoder.
- **DTDL [15]**: A dictionary-learning neural network model, which uses LSTM layers as the autoencoder.
- **DeepAR [7]**[1]: A popular time series prediction framework based on RNN. The main difference between DeepAR and Seq2Seq is that DeepAR introduces distributions as output at each timestep.
- **ConvTrans [12]**[2]: A model based on transformer [45] structure and LogSparse convolutional self-attention layers. LogSparse layer reduces the number of dot products per self-attention layer.

For the models that we failed to find open-source implementations, we tried to replicate the model structure in their paper to our best effort.

Most neural network models are based on either LSTM recurrent networks [44] or transformers [45]. For seq2seq models, the decoder input (output of former timestep) is concatenated with "known" variables; other networks already integrated static and extra sequences in their inputs.

In our experiments, we include both variations of TFTOps:

- **TFTOps**: The original TFT model, which is similar with [3].
- **TFTOps(prob)**: The TFT model with extra binned and probabilistic inputs.

C. EVALUATION

The effectiveness of the prediction model can be assessed from two different aspects. First, we want the prediction to be precise when the system runs normally. Second, we want

[1]https://github.com/jdb78/pytorch-forecasting
[2]https://github.com/mlpotter/Transformer_Time_Series

the prediction model to raise an alert when the system is in an erroneous state.

The first requirement is evaluated by computing the average L_1 distance between the following two metric values: the value predicted by the TFTOps model, and the real value in the timestamp on which the prediction was made.

Per timestamp i, the mean absolute error $MAE_(i)$ is defined as:

$$MAE(t+\tau) = |\hat{y}^{(t_0)}(t+\tau) - y(t+\tau)|$$

The final evaluation metric is the average over timesteps and sequences:

$$MAE = \frac{1}{n\tau_{\max}} \sum_{i=1}^{n} \sum_{\tau=1}^{\tau_{\max}} MAE_{(i)}(t+\tau)$$

Note that y is taken from the **normalized** sequence. The per-sequence loss $\sum_{\tau=1}^{\tau_{\max}} MAE_{(i)}(t+\tau)$ resembles the metric $MAE\%$ of this sequence, which is usually defined as

$$MAE\% = \frac{\sum_{\tau=1}^{\tau_{\max}} |A(\hat{y}^{(t_0)}(t+\tau)+B) - y'(t+\tau)|}{|\sum_{\tau=1}^{\tau_{\max}} |y'(t+\tau)}$$

where y' is the raw value retrieved from data source (without normalization) and A, B are two factors for reverting the normalization process.

We modified the output layer of those neural network models to enable quantile loss function. Different quantiles are trained in the same training loop, while their MAEs are cauculated separately.

The second requirement is evaluated by investigating the errorneous states in daily maintainence, or injecting exceptions into system and examine the supervised methods such as record and precision. We do not consider the F-metric due to the great difference between the number of normal and abnormal cases. Even after we injected exceptions in the system, abnormal cases were still extremely rare.

D. IMPLEMENTATION

This subsection describes the implementation details of TFTOps model in production envirionment.

1) AUTO UPDATE

The statistic of time series in a prouction system is always shifting over time. As a result, the predictior model must also have a updating routine.

In our production deployment, training and prediction are ran on different, independent processes. The update are scheduled once per week. We fetch the most recent data from databses to train TFTOps model; after training is done, the best model is saved to file system, and a message is send through RabbitMQ message queue. When the prediction process receives that message via polling, it will trigger a ‘‘reload’’ function which causes the prediction process to discard the old model and reload the new model from disk.

TABLE 4. Experiment results on PNE dataset.

Model	MAE(q=0.5)	MAE(q=0.1)	MAE(q=0.9)
ARIMA	7.6144	-	-
ETS	5.9987	16.5047	15.2615
MQRNN	1.8890	3.7035	1.8232
RoughAE	2.4901	2.3246	3.2497
DTDL	1.6633	1.9547	1.7837
DeepAR	1.0176	0.5384	1.3539
ConvTrans	1.2944	0.5322	1.7291
TFTOps	1.2706	0.9471	1.2419
TFTOps(prob)	**0.8397**	**0.5314**	**0.6932**

2) DATASET ERROR HANDLING

In production environments, each model corresponds to 2 weeks/one month of training data. It is nearly impossible for the microservice system to keep a consistent state for weeks. In our cases, Kubernetes pods (or their exporters) and Redis jobs are dynamically created and destroyed over time. As a result, we must deal with ‘‘dirty’’ data.

- Prometheus metrics: Prometheus node-exporters generate a metric (kube_node_status_condition) to track the state of each node. As we wish to model the system's normal and steady state, the metrics generated while kube_node_status_condition {condition='Ready', status='true'} $\neq$ 1 (which means the node is not ready) are discarded from the training set.
- Redis Connect: For each Redis job, other than inferring the mean and variance directly from the training set, we observe a longer range (6 months) to retrieve a better estimation of mean and variance. If the job was started in 6 months, the observation becomes its full lifespan. Besides rescaling the input/output, we also use this observation to rule out outliers. Outliers are replaced with $\mu \pm 3\sigma$. In our systems, the number of connections per Redis job is tracked by Kafka messages. A scheduled producer fetches the status of Redis clusters and uploads them to the given topic. Both fetch and upload procedures are prone to errors during network fluctuations. Therefore, we observed missing data points occasionally. When the number of consecutive missing observations is relatively small ($\leq$ 5, 5 minutes), we assume the system is running normally. In this case, we use the latest visible observation for filling in the missing values. Therefore, our pre-processing step can generate training data generated across the gap. On the contrary, when a wider gap ($>$ 5) appears, we assume that the system's state is questionable, thus discarding the missing observations.

E. RESULTS: EVALUATION ON TEST DATASETS

The MAE results of models mentioned in previous section are shown in Table 4 and Table 5.

Generally speaking, the performance of modern models is better than traditional ARIMA/ETS models. It is worth noticing that neural network models are especially good at

TABLE 5. Experiment results on RDC dataset.

Model	MAE(q=0.5)	MAE(q=0.1)	MAE(q=0.9)
ARIMA	38.8198	-	-
ETS	23.9378	70.3496	71.7681
MQRNN	24.3383	29.4822	20.6325
RoughAE	31.6767	41.2920	37.3145
DTDL	28.7762	15.0396	30.3068
DeepAR	25.7791	24.0882	22.1861
ConvTrans	26.5045	20.8684	21.8974
TFTOps	**7.4489**	10.1459	5.9466
TFTOps(prob)	9.1080	**7.4725**	**3.7677**

predicting quantiles: all the neural network models' quantile losses(q=0.1 and q=0.9) are far better than that of ETS.

Among all of neural network models, we can conduct that TFTOps achieves excellent MAE in both datasets, exceeding other SOTA models, and is especially good for the RDC dataset which keeps oscillating.

The probabilistic input scheme for TFTOps also have positive effect on MAE; however, this scheme introduces some extra parameters and requires slightly more computations in preprocessing and embedding extraction phases. However, due to the probabilistic features are vectorized and fed into the variable selection layer, the extra cost is trivial.

F. RESULTS: THE ROBUSTNESS OF TFTOPS AGAINST NOISE

When building a metric prediction system in real world, the developers have a variety of static or variable features to choose from. For example, when predicting the CPU usage, one can easily acquire the following features from many sources, especially from prometheus node-exporter:

- CPU model, generation and performance benchmarks (static)
- The number of processes running on the same machine (variable)
- The network load of the same machine (variable)
- KPI of related services (variable) ...

Feature without impacts are considered "noise" and is harmful to the model. Generally, one would run a grid-search to validate the effectiveness of each feature and find the optimal set of features. However, training all sorts of deep-learning models require a non-trivial amount of time. To metigate this problem, the model itself should be robust against the noise.

We designed a new dataset PNENoise based on PNE to test different model's ability to filter noise. In PNENoise, new artificial noise features are concatenated to each row. Those "noise" features are sampled from following distributions to reflect various situations:

- Uniform: a uniform distribution between [0,1].
- Normal: a normal distribution $N(0, 1)$.
- Periodic normal: a normal distribution whose centroid is a function of time t. This reflects some periodic component. We chose $N(5sin(t/24), 1)$.
- Random category: a randomly assigned "class" from $0, 1, 2, 3, 4$. This is a static feature.

TABLE 6. Results on PNENoise dataset.

Model	Original	Noise	MAE%
MQRNN	1.8890	2.5695	36.02%
RoughAE	2.4901	2.5428	6.13%
DTDL	1.6633	1.8148	9.10%
DeepAR	1.0176	1.3679	34.42%
ConvTrans	1.2944	1.6581	28.10%
TFTOps	1.2706	1.3531	6.49%
TFTOps(Prob)	0.8397	0.9034	7.58%

To evaluate the robustness, we re-train different models (except ARIMA and ETS) on the PNENoise dataset from scratch and compare their performance on the MAE($q = 0.5$) metric. The increased percentage on MAE($q = 0.5$) quantifies the model's robustness against noise; a lesser increase means better robustness. Note that, we use the same hyperparameter search space when training the models on PNENoise.

Results are shown in Table 6. It can be conducted that, traditional neural-network based methods(MQRNN,DeepAR and ConvTrans) suffers from noisy features, while RoughAE/DTDL and TFTOps are only slightly affected. The implementation of RoughAE and DTDL directly addressed the robustness problem and solving it by model designations such as rough inputs [14] and dictionary learning [15], thus achieved better robustness. However, their performances are slightly worse than deep neural networks. The TFTOps model, on the other hand, maintains robustness while enhancing its performance.

G. RESULTS: EVALUATION ON REAL ENVIRONMENT

As mentioned in [7], the quantile loss function and outputs naturally becomes a good detector of anomalies. Every time the real observed target violates the prediction (a target value that is even higher than the upper quantile or lower than the lower quantile) indicates that the sequence is in an abnormal state.

We performed the evaluation on the production environment which generates RDC dataset. The TFTOps model was kept serving for a month. During that period, the operators recorded 17 abnormal incidents. When TFTOps model reports an anomaly, we check that whether an abnormal incident happens within a 10-minute range (true positive) or not(false positive).

The confusion matrix and evaluation result is shown in table 7 and table 8.

We can conclude that the model successfully extinguishes most of anomalies at a cost of relatively low precision.

Shifting the quantile loss (q=0.1/q=0.9) towards 0/1 should improve precision while lowering recall. The selection of quantile will greatly influence the performance of model in production schemes. A practical solution is to predict many quantiles(for example, q=[0.05, 0.1, 0.15...,0.95]), and select a proper quantile as detector of anomalies according to real situations.

To illustrate the conclusion above, we re-trained the model using the same dataset and hyperparameters, only altered

TABLE 7. confusion matrix of RDC system evaluation.

		Redis Prediction outcome		
		p	n	total
actual value	p′	True Pos. = 13	False Neg. = 4	$P' = 17$
	n′	False Pos. = 27	True Neg. = 43156	$N' = 43183$
	total	$P = 40$	$N = 43160$	

TABLE 8. Production results on RDC system.

metric	value
recall	76.4%
precision	32.5%
F1-score	0.4561

TABLE 9. Confusion matrix of new quantiles for RDC.

		Redis Prediction outcome		
		p	n	total
actual value	p′	True Pos. = 11	False Neg. = 6	$P' = 17$
	n′	False Pos. = 20	True Neg. = 43156	$N' = 43163$
	total	$P = 31$	$N = 43169$	

TABLE 10. Production results on RDC(new quantiles).

metric	value
recall	64.7%
precision	35.5%
F1-score	0.4583

the quantiles to $q = [0.05, 0.5, 0.95]$. The new model's confusion matrix and metrics are shown in table 9 and table 10

Shifting the quantile loss (q=0.1/q=0.9) towards 0/1 should improve precision while lowering recall. The selection of quantile will greatly influence the performance of model in production schemes. A practical solution is to predict many quantiles(for example, q=[0.05, 0.1, 0.15…,0.95]), and select a proper quantile as detector of anomalies according to real situations.

Our investigation shows that in our production setting, human operators slightly favor recall over precision. Therefore, we chose $q = [0.1, 0.5, 0.9]$ even when other choice ([0.05, 0.5, 0.9]) has a higher F1 score.

More generally, the choice between recall and precision is determined by the nature of the system that generates our metrics. When it is easy to validate whether an actual error occurs, or every occurence of the error tends to have critical consequences, operators favor recall over precision; when the error can be resolved automatically, operators tend to favor precision over recall to eliminate false positives.

TABLE 11. Efficiency on prediction stage.

Dataset	$\#Seq$	$l_{encoder}$	$l_{decoder}$	Avg. time(s)
PNE	528	48	24	55.23±5.62
RDC	34	50	10	4.55±1.01

H. EFFICIENCY

The time cost on prediction stage of TFTOps model is listed in table 11.

The column "Avg. time" is the average time to process a slice of data, which contains metrics generated by the whole system ($\#Seq * l_{encoder}$), and the model should make ($\#Seq * l_{decoder}$) predictions.

When a forward pass is run, the data flows through an LSTM layer and a stack of self-attention layers. The time complexity ties closely with the total sequence length of encoder and decoder, $l_e + l_d$. For LSTM, its complexity is $O(l_e + l_d)$, which is ruled out by self-attention layer's complexity $O((l_e + l_d)^2)$. Depending on the degree of parallelism, the LSTM layer generally costs more time when $(l_e + l_d)$ is relatively small, which is our case. Note that, except the LSTM layer, the model is purely feed-forward (the result generated by LSTM does not feedback into previous layers), thus the performance of pre-processing and output dense layers are theoretically better than LSTM; In the prediction stage, the prediction time is roughly linear with number of sequences in the system (n = #Seq). Thus, the overall complexity is $O(n(l_e + l_d))$(for shorter sequences) or $O(n(l_e + l_d)^2)$(for longer sequences).

In the PNE dataset task, we retrieve data every 5 minutes; in RDC dataset task, we retrieve data once per minute. According to table 11, in both cases, the efficiency of TFTOps model meets our production requirements.

V. CONCLUSION

We proposed TFTOps, a variant of Temporal Fusion Transformer [3] designed for AIOps unsupervised anomaly detection tasks. We also improve TFTOps by introducing probabilistic inputs, which further boosts the accuracy of TFTOps in our experiments. Our findings prove that the TFT model concept is well-suited for a modern multi-metric monitoring setup, satisfying requirements on both accuracy and efficiency.

The merits of TFTOps includes:

- Flexiblity. User can include various features (real-valued / categorical / probabilistic, variable / static) in their hypothesis. TFTOps provides an elegant way to blend the features into its training and prediction process.
- Robustness. If the new feature is proven to be noise, the TFTOps model has stronger robustness compared to other neural network models. Therefore, it shortens the time-consuming scheme of feature selection. With little

effort, the operators can generate valuable predictions with the off-the-shelf TFTOps model.

VI. FUTURE WORKS

In our experiments, we treat each sequence independently (except they share some categorical static inputs). Intuitively, in k8s or other cluster settings, introducing features from "related" nodes would be beneficial for the accuracy of our model. Besides, other possible data sources, such as embedded system logs, can be used as a feature. We leave both directions for future works.

APPENDIX

A. HYPERPARAMETERS

We use the following grid search scheme for the optimal hyperparameters in experiments on both PNE and RDC datasets.

The parameter name, search space and effect of hyperparameters are listed below:

- Dropout rate: {0.1, 0.2}. This parameter controls probability of dropout in GLU layer before gating layer and layer normalization.
- Hidden layer size: {4, 8, 16, 32}. This controls the size of vector input/output of hidden layer(d_{model}).
- Learning rate: {0.1, 0.01, 0.001}. Fine-grained tuning can be conducted according to dataset to further improve the result.
- #heads: {4,6,8}. The number of heads in multi-head attention layers.
- Stack size: {1,2,3,4}. the number of stacking self-attention (SA) layers.

For the PNE dataset, optimal hyperparameters are:

- Dropout rate: 0.2
- Learning rate: 0.001
- Hidden layer size: 8
- #heads: 4
- Stack size: 4

For the RDC dataset, optimal hyperparameters are:

- Dropout rate: 0.1
- Learning rate: 0.01
- Hidden layer size: 8
- #heads: 4
- Stack size: 2

According to our observations, both datasets does not require a large embedding/feature space. On the other hand, they do need to stack some self-attention layers to achieve the optimal performance. Intuitively, higher dimension of embeddings means we can extract rich information from input features(static and variable). However, for our dataset, the inter-relationship among different timesteps of the metric itself seems to be more important, which is mainly modeled by the LSTM and Transformer layers.

REFERENCES

[1] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng, J. Chen, Z. Wang, and H. Qiao, "Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications," in *Proceedings of the 2018 World Wide Web Conference*, ser. WWW '18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2018, pp. 187–196, doi: 10.1145/3178876.3185996.

[2] S. Aghabozorgi, A. S. Shirkhorshidi, and T. Ying Wah, "Time-series clustering—A decade review," *Inf. Syst.*, vol. 53, pp. 16–38, Oct. 2015.

[3] B. Lim, S. Ö. Arık, N. Loeff, and T. Pfister, "Temporal fusion transformers for interpretable multi-horizon time series forecasting," *Int. J. Forecasting*, vol. 37, no. 4, pp. 1748–1764, Oct. 2021.

[4] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control*. Hoboken, NJ, USA: Wiley, 2015.

[5] R. J. Hyndman and G. Athanasopoulos, *Forecasting: Principles and Practice*. Melbourne, VIC, Australia: OTexts, 2018.

[6] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. 27th Int. Conf. Neural Inf. Process. Syst.*, vol. 27, 2014, pp. 3104–3112.

[7] D. Salinas, V. Flunkert, J. Gasthaus, and T. Januschowski, "DeepAR: Probabilistic forecasting with autoregressive recurrent networks," *Int. J. Forecasting*, vol. 36, no. 3, pp. 1181–1191, Jul. 2020.

[8] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[9] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, vol. abs/1412.3555, 2014. [Online]. Available: http://arxiv.org/abs/1412.3555

[10] S. S. Rangapuram, M. W. Seeger, J. Gasthaus, L. Stella, Y. Wang, and T. Januschowski, "Deep state space models for time series forecasting," in *Proc. Adv. Neural Inf. Process. Syst., Annu. Conf. Neural Inf. Process. Syst. (NeurIPS)*, vol. 31, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Montréal, QC, Canada, Dec. 2018, pp. 7796–7805. [Online]. Available: https://proceedings.neurips.cc/paper/2018/hash/5cf68969fb67aa6082363a6d4e6468e2-Abstract.html

[11] D. P. Kingma and M. Welling, "Auto-encoding variational Bayes," 2020, *arXiv:1312.6114*.

[12] S. Li, X. Jin, Y. Xuan, X. Zhou, W. Chen, Y.-X. Wang, and X. Yan, "Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting," in *Proc. Adv. Neural Inf. Process. Syst., Annu. Conf. Neural Inf. Processing Syst. (NeurIPS)*, vol. 32, Vancouver, BC, Canada, Dec. 2019, pp. 5244–5254. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/6775a0635c302542da2c32aa19d86be0-Abstract.html

[13] C. Fan, Y. Zhang, Y. Pan, X. Li, C. Zhang, R. Yuan, D. Wu, W. Wang, J. Pei, and H. Huang, "Multi-horizon time series forecasting with temporal attention learning," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2019, pp. 2527–2535.

[14] M. Khodayar, O. Kaynak, and M. E. Khodayar, "Rough deep neural architecture for short-term wind speed forecasting," *IEEE Trans. Ind. Informat.*, vol. 13, no. 6, pp. 2770–2779, Dec. 2017.

[15] M. Khodayar, J. Wang, and Z. Wang, "Energy disaggregation via deep temporal dictionary learning," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 31, no. 5, pp. 1696–1709, May 2020.

[16] R. Wen, K. Torkkola, B. Narayanaswamy, and D. Madeka, "A multi-horizon quantile recurrent forecaster," 2017, *arXiv:1711.11053*.

[17] F. Ayed, L. Stella, T. Januschowski, and J. Gasthaus, "Anomaly detection at scale: The case for deep distributional time series models," in *Proc. Int. Conf. Service-Oriented Comput.* Cham, Switzerland: Springer, 2020, pp. 97–109.

[18] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya, "Anomaly detection using program control flow graph mining from execution logs," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 215–224.

[19] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, and D. Pei, "Unsupervised detection of microservice trace anomalies through service-level deep Bayesian networks," in *Proc. IEEE 31st Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2020, pp. 48–58.

[20] J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey," *ACM Comput. Surv.*, vol. 55, no. 3, pp. 1–39, Mar. 2023.

[21] B. Rabenstein and J. Volz, *Prometheus: A Next-Generation Monitoring System (Talk)*. Dublin, Republic of Ireland: USENIX Association, May 2015.

[22] *Grafana: The Open-Source Platform for Monitoring and Observability*. Accessed: Jan. 15, 2024. [Online]. Available: https://github.com/grafana/grafana

[23] Prometheus. *Querying Basics: Prometheus*. Accessed: Jan. 15, 2024. [Online]. Available: https://prometheus.io/docs/prometheus/latest/querying/basics/

[24] S. Fu, "Performance metric selection for autonomic anomaly detection on cloud computing systems," in *Proc. IEEE Global Telecommun. Conf. (GLOBECOM)*, Dec. 2011, pp. 1–5.

[25] G. Jung, G. Swint, J. Parekh, C. Pu, and A. Sahai, "Detecting bottleneck in n-tier it applications through analysis," in *Proc. Int. Workshop Distrib. Syst., Oper. Manage.* Cham, Switzerland: Springer, 2006, pp. 149–160.

[26] J. Parekh, G. Jung, G. Swint, C. Pu, and A. Sahai, "Issues in bottleneck detection in multi-tier enterprise applications," in *Proc. 14th IEEE Int. Workshop Quality Service*, Jun. 2006, pp. 302–303.

[27] Y. Tan, *Online Performance Anomaly Prediction and Prevention for Complex Distributed Systems*. Raleigh, NC, USA: North Carolina State Univ., 2012.

[28] X. Gu and H. Wang, "Online anomaly prediction for robust cluster systems," in *Proc. IEEE 25th Int. Conf. Data Eng.*, Mar. 2009, pp. 1000–1011.

[29] R. Powers, M. Goldszmidt, and I. Cohen, "Short term performance forecasting in enterprise systems," in *Proc. 11th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2005, pp. 801–807.

[30] T. Wang, W. Zhang, J. Wei, and H. Zhong, "Workload-aware online anomaly detection in enterprise applications with local outlier factor," in *Proc. IEEE 36th Annu. Comput. Softw. Appl. Conf.*, Jul. 2012, pp. 25–34.

[31] D. J. Dean, H. Nguyen, and X. Gu, "UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *Proc. 9th Int. Conf. Autonomic Comput.*, Sep. 2012, pp. 191–200.

[32] R. J. Hyndman, E. Wang, and N. Laptev, "Large-scale unusual time series detection," in *Proc. IEEE Int. Conf. Data Mining Workshop (ICDMW)*, Nov. 2015, pp. 1616–1619.

[33] C. Monni, M. Pezzè, and G. Prisco, "An RBM anomaly detector for the cloud," in *Proc. 12th IEEE Conf. Softw. Test., Validation Verification (ICST)*, Apr. 2019, pp. 148–159.

[34] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, Jul. 2006.

[35] T.-Y. Liu, "Learning to rank for information retrieval," *Found. Trends Inf. Retr.*, vol. 3, no. 3, pp. 225–331, 2009.

[36] Y. Liang, S. Ke, J. Zhang, X. Yi, and Y. Zheng, "GeoMAN: Multi-level attention networks for geo-sensory time series prediction," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, Jul. 2018, pp. 3428–3434.

[37] Y. Liu, C. Gong, L. Yang, and Y. Chen, "DSTP-RNN: A dual-stage two-phase attention-based recurrent neural network for long-term and multivariate time series prediction," *Expert Syst. Appl.*, vol. 143, Apr. 2020, Art. no. 113082, doi: 10.1016/j.eswa.2019.113082.

[38] C. Wang, K. Wu, T. Zhou, G. Yu, and Z. Cai, "TSAGen: Synthetic time series generation for KPI anomaly detection," *IEEE Trans. Netw. Service Manage.*, vol. 19, no. 1, pp. 130–145, Mar. 2022.

[39] H. Ren, B. Xu, Y. Wang, C. Yi, C. Huang, X. Kou, T. Xing, M. Yang, J. Tong, and Q. Zhang, "Time-series anomaly detection service at Microsoft," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2019, pp. 3009–3017.

[40] H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, and W. Zhang, "Informer: Beyond efficient transformer for long sequence time-series forecasting," in *Proc. AAAI Conf. Artif. Intell.*, vol. 35, no. 12, May 2021, pp. 11106–11115.

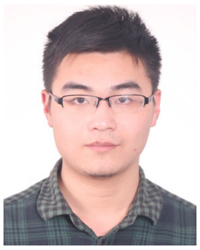
[41] J. Xu et al., "Autoformer: Decomposition transformers with auto-correlation for long-term series forecasting," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 34, 2021, pp. 22419–22430.

[42] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (ELUs)," 2015, *arXiv:1511.07289*.

[43] C. Favorita. (2018). *Corporacion Favorita Grocery Sales Forecasting Competition*. [Online]. Available: https://www.kaggle.com/c/favorita-grocery-sales-forecasting/

[44] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, May 2013, pp. 6645–6649.

[45] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst., Annu. Conf. Neural Inf. Process. Syst.*, vol. 30, Long Beach, CA, USA, Dec. 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html



HAORAN LUO (Member, IEEE) was born in Fujian, China, in 1996. He received the B.S. and M.S. degrees in computer science from Shanghai Jiao Tong University, Shanghai, China, in 2018 and 2021, respectively. From 2018 to 2019, he was an Intern with Bosch (China) Investment Company Ltd. From 2019 to 2020, he was an Intern with the Algorithm Department, 360 Digitech Inc., Shanghai. He has been a Researcher with the China Telecom Research Institute, Guangzhou, China, since 2021. His research interests include machine learning, devOps, AIOps, and natural language processing.

YONGKUN ZHENG was born in Shantou, Guangdong, China, in 1990. He received the M.S. degree in software engineering from Southeast University, in 2016. From 2014 to 2015, he was an Intern with the Cloud Computing Center, Chinese Academy of Sciences, engaged in research on the application of big data in remote sensing technology. In 2015, he was an Intern with Baidu, engaged in data analysis. Since 2016, he has been a Researcher with the China Telecom Research Institute, Guangzhou, China. His main research interests include AIOps, devOps, and big data analytics.

KANG CHEN (Member, IEEE) was born in Anhui, China, in 1972. He received the B.S. degree from Nanjing University, Nanjing, China, in 1993, and the M.S. degree in computer science from Jinan University, Guangzhou, China, in 1999. Since 1999, he has been a Researcher with the China Telecom Research Institute, Guangzhou. His research interests include big data and machine learning.

SHUO ZHAO was born in Jilin, China, in 1988. He received the B.S. degree in computer science and technology and the M.S. degree in computer science from the Beijing University of Posts and Telecommunications, Beijing, China, in 2011 and 2014, respectively. Since 2014, he has been an IT Engineer with China Telecom Corporation Ltd., Beijing. His research interests include machine learning and AIOps.