Deep Learning Models for Automated Identification of Scheduling Policies

Yichong Chen
Imperial College London, UK
yichong.chen119@imperial.ac.uk

Giuliano Casale
Imperial College London, UK
g.casale@imperial.ac.uk

***Abstract*—Queueing network models are commonly used as performance models of distributed software applications and service-based systems. Although several methods exist for learning their parameters, such as demand estimation methods, little research has been carried out in the literature on automatically identifying scheduling policies from empirical datasets. Scheduling policies and their parameters have an impact on the model's stationary distribution in general, thus their correct determination is important for model accuracy. They are particularly relevant for correctly estimating percentiles and higher-order moments of performance indexes such as response times. We propose a deep learning technique based on transformer models - a common technique in natural language processing, to address the lack of methods for this parameter identification problem. From a sample path of the joint network state, or an aggregate thereof, our approach can classify the scheduling policy of the stations in a queueing network. We show that the transformer model delivers good-classification precision and recall, improving significantly over support vector machines or simpler recurrent neural networks.**

I. Introduction

Queueing network models are a type of stochastic model that is commonly used to simulate congestion in software applications and service-based systems [1]. The identification of artificial intelligence and machine learning (AI/ML) methods that can help to automatically identify a model from empirical data is a natural question that arises in conjunction with the current growth of AI technology.

Over the last decade, a comprehensive range of studies have investigated estimating service demands, which are important drivers of model prediction accuracy [2] and one of the main challenges in the performance model identification field. For example, in [3], the authors apply maximum likelihood estimation to predict demands in closed queueing networks. Similarly, the work in [4] tackles the same problem via Gibbs sampling and Monte Carlo integration [5]. Prior art in parameter estimation also includes open models, such as [6] that estimates service and arrival rates limiting distribution based on the number of jobs in the node. AI/ML are also used in some prior arts. For example, in [7], the authors apply a fluid approximation to the queueing network [8] to model the system dynamics and use a recurrent neural network to predict the queue length and encode the transition probability matrix and the service rate of the queueing network. The work in [9] predict the waiting time of jobs in a queueing system with a Gaussian mixture model in which parameters are generated from a fully-connected neural network. The applicability of these methods often depends on the precise assumptions of the scheduling policy at play, making it critical to identify the scheduling policy in use inside a system with high certainty. For example, [10] proposes various algorithms for estimating missing data inside a finite sample path of a queueing network, depending on the scheduling policy. Therefore, the application of this body of work on demand estimation effectively requires to know the scheduling policy at every node before applying the proposed algorithm to the dataset, motivating the use of specialized methods to learn such scheduling characteristics from measured data.

The widespread diffusion of cloud-hosted services makes model classification useful for infrastructure providers to estimate the topology and resource consumption of an application from passive measurements (e.g., network packet sniffing), is another motivator for the scheduling identification problem we consider. Learning scheduling policies in this setting may in principle help in better understanding the scheduling characteristics of a distributed system, supporting performance prediction.

Motivated by the above considerations, this paper proposes a transformer model [11] to automatically identifying the scheduling policies of a network of resources, modeled as queueing stations, assuming that partial measurements are obtained for their state. To the best of our knowledge, the present work is the first one to develop an AI method for learning scheduling characteristics from partial monitoring data and accounting for the presence of queueing phenomena. We show that transformer models, commonly used in natural language translation problems, can be effectively adapted to the queueing model identification problem under study. Experiments with simulation data show that it is possible to recover with relatively good accuracy the scheduling characteristics of the system that generated the data using a supervised learning approach. The proposed method can result in accuracy and F1-score that both exceed 0.80. Satisfactory accuracy is observed also in sensitivity analysis experiments, where we reduce the amount of information available to the method, or include job priorities in the model.

The paper is organized as follows: Section 2 introduces the classes of models studied throughout this paper. Section 3 overviews certain AI/ML models applicable to the problem at hand. Section 4 introduces the proposed transformer model.

TABLE I
STATE REPRESENTATION FOR A GIVEN SINGLE-SERVER STATION

Scheduling	*State vector*	*Condition*
FCFS, LCFS, HOL	$(c_b, \ldots, c_1, s_1, \ldots, s_R)$	$\sum_r s_r = 1$
PS	$(s_1, \ldots, s_R)$	None
SIRO	$(b_1, \ldots, b_R, s_1, \ldots, s_R)$	$\sum_r s_r = 1$

Section 5 gives experimental results, followed by conclusions.

II. BACKGROUND

Queueing network models have been extensively studied in the scientific literature and applied to many computer and communication systems [12]. According to certain statistical distribution, service times have a random duration. Jobs may either visit the system and then leave (open classes), or perpetually cycle in the network (closed classes). Routing is probabilistic and governed by a discrete-time Markov chain.

At each station, a scheduling policy determines which job will be selected for processing when a server is idle. Some of the most commonly used scheduling policies include first-come first-served (FCFS), last-come first served (LCFS), processor sharing (PS), and service in random order (SIRO) [1]. If jobs are assigned a static priority, the FCFS rule is typically replaced by the Head-of-Line (HOL) policy, in which the jobs with the highest priority are served first and then in order of arrival within that group.

Let us consider a model with M stations and where jobs can belong to one of R service classes. Class switching is not allowed and service times are assumed to be exponentially distributed. When we represent such systems as a continuous-time Markov chain, the state vector for a given single-server station can be represented as in Table I, based on the following symbols:

- b number of jobs in the waiting buffer in the current state.
- c_j: class of job waiting at the jth position in the waiting buffer.
- n_r: Number of class-r jobs in the station.
- b_r: Number of class-r jobs in the buffer.
- s_{rk}: Number of class r jobs running in the server.

It would be fairly simply to classify scheduling by observing the random variables shown in Table I. The problem we consider here is however to do so by observing aggregate state vectors

$$x(t) = (x_{11}(t), x_{12}(t), \ldots, x_{1R}(t), \ldots, x_{MR}(t)) \quad (1)$$

where $x_{ij}(t)$ is the random variable denoting the number of jobs in class j at station i at the measurement time t. The sample path of aggregate states has a more compact representation than using the exact one in Table I. It is also easier to collect and monitor in real-world systems in practice, since it does not require to track the state of individual jobs, but rather how any jobs for each class are inside the system. Thus, it is a quantity that can be estimated by input-output measurements of the system. A drawback is that (1) also implies that two models with different scheduling policies may produce the same sample routes in terms of aggregate variables, driving our research into learning methods to address its ambiguity. That is, we take as our input dataset, for the AI classifiers studied later, a sample path of aggregate states $x(t)$, where time t belongs to a finite set of timestamps T at which we have collected observations. Our goal is to classify the scheduling policy based on the finite realization.

III. SEQUENCE CLASSIFICATION METHODS

Classifying the scheduling policy in a single-server station is easy if the state space is small and the station states can be continuously observed. However, in many real systems, such as in a web server, hundreds of thousands of requests received in a matter of seconds, making precise tracking of the system state costly. Indeed, tracing methods for distributed systems are forced for this very same reason to adopt sampling methods. Besides, the system may leverage resources across a cluster and thus require models with multiple queueing stations. Therefore, collecting all the states at every node in a reliable fashion becomes more difficult. Furthermore, some data may be absent, further complicating the process.

Secondly, in the presence of aggregation, it is difficult to discern distinct traits that distinguish policies from sample paths. In our experiments, we discovered that samples paths across policies appear appear qualitatively comparable when inspected visually. Faced with this challenge, we propose the use of AI/ML methods in solving the model identification problem..

In this section, we introduce three possible architectures that may be used for this problem, leveraging SVMs [13], RNNs [14], and transformers [11]. As our experimental validation later demonstrates, the transformer outperforms the other two techniques, we expand on this in Section 4, since it is relatively little understood within the performance evaluation community.

A. Support Vector Machines

The support vector machine (SVM) is a well-established machine learning model introduced in [13]. The SVM draws a linear decision boundary which maximize the gap between two classes so that to separate two classes. The model may fail if the data are not linearly separable. We can address this by projecting the data to a higher dimension where the data are linearly separable with a chosen kernel function. The radial basis function (RBF) is commonly used in classification problems and also used throughout the present work.

To apply SVM to time series data, we consider the states of a station throughout a period as input data, which may be seen as a matrix of size $n \times m$ where n is the sequence length and m is the dimension of the aggregate state vector. This matrix is transformed to an input vector of size nm. Thus, SVMs can be applied to time series data by increasing the number of input features.

B. Recurrent Neural Networks

In deep learning (DL) and natural language processing (NLP), recurrent neural networks (RNN) are often used to

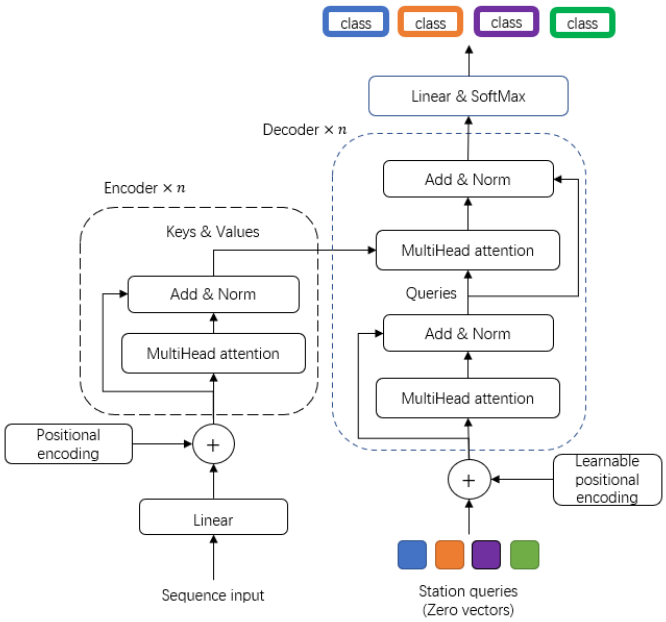


Fig. 1. Proposed Transformer architecture

learn patterns within a time series. The long-short term memory (LSTM) mechanism is used to prevent the vanishing gradient problem [14] . To produce a classification decision, the RNN outputs from a trace need to be condensed to a single vector. This may either be done by averaging all outputs, or only by considering the output associated to the last state. A linear layer then receives the compressed vectors and convert them to the required size.

The model may be generalized so that even if the number of stations in different queueing networks used for training or testing differs, we are still able to classify scheduling with a single model that only has a bound on the maximum number of stations that the network can include. Let us define the maximum possible station in the training data as n_{query}. The output size of the linear layer becomes $(B, n_{query} \times C)$ where C is the number of classes. Finally, the output is reshaped to (B, n_{query}, C) so that model can be trained in a supervised fashion.

C. Transformer model

The transformer model [11] has made outstanding achievements on solving NLP problems like machine translation [15] and language model construction [16]. To apply this model to the problem at hand, we introduce a modified transformer architecture as shown in Figure 1; details of this architecture are described later in the next section.

IV. Background on Transformer Model Architecture

A. Overview

The architecture seen in Figure 1 is described in detail in this section. The meaning of the blocks in the model architecture is as follows. The transformer model is an encoder-decoder model. The encoder maps the state sequence to a new sequence of representations that is passed as one of the input of the decoder. The decoder then generates the output with n_{query} station queries that identify the scheduling characteristics. These queries are initially zero vectors that are modified by the decoder as they pass through the blocks shown in Figure 1. Therein, the positional encoding blocks embed the order of the sequence and the multi-head attention blocks process the correlation between each states within a sequence. Finally, using a linear feed-forward network and the softmax function, the data is processed so that the output of the model represents the probability of a given scheduling policy being used at each station.

B. Attention

The attention mechanism is the crucial part of the transformer model. We here review the essential concepts of the application of the method in [11] to the queueing domain. This mechanism can process the relation between any two states in the sequence, and its output is a vector $a \in \mathbb{R}^{n \times d_v}$, where d_v is a parameter denoting the dimension of the key vectors. This layer is defined on the *attention function*, which is a function that delivers a high contribution to the output of the corresponding query if two states are highly correlated, and vice-versa if their correlation is negligible. The attention function takes three inputs, which are queries and key-value pairs, to produce the output. The contents of the key-value pairs are described in the next sub-sections, we here focus on the definition of attention and for the moment these may be treated as arbitrary.

In more details, suppose the queries Q and keys K are in $\mathbb{R}^{n \times d_k}$, and the values V is in $\mathbb{R}^{n \times d_v}$ where n is the number of states in the trace. The equation for calculating the output of the attention function is as follows

$$a = \mathrm{A}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{2}$$

From this we can compute the correlation between the query and the key. More precisely, this is used in the transformed model to compute the correlation between any two states x_i and x_j.

In our model, each layer contains multiple attention functions. Each attention function is referred to as *attention head* and it concentrates on different features of the sequence. These attention heads form the multi-head attention layer which is written as follows [11],

$$\mathrm{MH}(Q, K, V) = \mathrm{Concat}(a_1, \ldots, a_h)W^O \tag{3}$$

$$a_i = \mathrm{A}(QW_i^Q, KW_i^K, VW_i^V), \quad i = 1, \ldots, h. \tag{4}$$

That is, in order to have different attentions, we project the queries and key-value pairs linearly for h times with learnable parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$ and $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$, where $d_k = d_v = d_{model}/h$, so that the projected queries and key-values pairs contain different

information that the attention function should focus on. All the attention outputs are concatenated together. To be more specific, if we take the attention outputs $a_i \in \mathbb{R}^{n\times d_v}$, all the matrices are stacked together to form a large matrix with more columns so that the shape of the concatenated matrix is (n, hd_v). The output is again projected with $W^O \in \mathbb{R}^{hd_v\times d_{model}}$ to process the output of different attention heads and get the final result.

C. Transformer Encoder-Decoder

Suppose the input state sequence is $x \in \mathbb{R}^{n\times d}$ where n is the number of states, and d is the dimension of a single state. The input is first projected to a higher dimension with a feed-forward network (FFN), so that $x' \in \mathbb{R}^{n\times d_{model}}$. In our experiment, adding this FFN can significantly improve the performance of the model.

Unlike the RNN model which inherently takes sequence order into account, the encoder and decoder of the transformer model only contain the multi-head attention and the FFN, neither of these networks takes sequence order into account. As a result, a positional encoding P is added to the input to embed the order of the sequence. In [11], the sine and cosine functions are used as the positional encoding. The functions are defined as follow

$$P_{t,2i} = \sin\left(\frac{t}{10000^{\frac{2i}{d_{model}}}}\right) \tag{5}$$

$$P_{t,2i+1} = \cos\left(\frac{t}{10000^{\frac{2i}{d_{model}}}}\right), \tag{6}$$

where $t = 0, 1, \ldots, (n-1)$. The positional encoding matrix P determined not by the input values but by the d_{model} and the position. Therefore, the order of a state can be represented by a unique vector P_t and add to the projected state vector x'_t (the row vector of x'), so that the input contains the information of the order.

In the encoder, the encoded state vectors become the queries and key-value pairs of the multi-head attention network. The output of the multi-head attention network is added with the input of the attention network, so that the loss can pass through the attention network or directly back-propagate to the previous layer [17]. Finally, in order to accelerate the training and minimize the covariate shift, we apply batch normalization to the resulted output [18].

In machine translation, the decoder of the transformer model takes the output of the encoder and the output of the previous prediction of the model. However, in our model, we classify the scheduling policy of all the stations simultaneously. To address this problem we use a matrix z of zeros with shape (n_{query}, d_{model}) as one of the inputs of the decoder. Since the model produces n_{query} different predictions, the input must be different before sending it into the decoder. As a result, a learned positional encoding is applied, and the input is combined with the positional encoding, just like in the encoder.

In the decoder, the multi-head attention network takes the output of the previous layer as the query and the output of the encoder as the key-value pairs so that the network considers both the input of the decoder and the output of the encoder.

D. Prediction FFN and loss function

In the final step, the prediction is computed with an FFN with softmax as the activation function. By using the softmax function, the probability of each scheduling policy used in the station can be computed.

To measure the performance of the model, we use cross-entropy (7) as the loss function. As we mention above, we set $M \leq n_{query}$ and use the "empty" label to represent the unused stations. The amount of "empty" labels can be larger than the other scheduling policies, so the model is easier to learn to classify the "empty" label. However, we hope that the model can also perform well on other policies. As a result, the loss of each class needed to be weighted. The empty class should have a significantly lower weight than the others. So the loss function is

$$\text{Loss}(\text{output}, y) = -\sum_{i=1}^{N}\sum_{j=1}^{C} \text{Weight}(y_{ij})(y_{ij}\log(\text{output}_{ij})) \tag{7}$$

where y_{ij} is the true scheduling policy of sample i at station j and output_{ij} is the corresponding predicted probability of using policy y_{ij}. $\text{Weight}(y_{ij})$ is the weight of policy y_{ij}. We need to find a model to minimize the cross-entropy in order to maximize the probability of giving the correct classification. We employ Adam [19] in our experiments.

V. Experiment

A. Data collection

A large number of sample paths of a closed network has been used to train the considered prediction models. We have used traces sampled from stochastic simulation of the continuous-time Markov chain underpinning the queueing model. The sampling has been performed using the MATLAB toolbox LINE [20], [21]. The queueing network routing matrices are randomly generated. Randomized parameters include the number of nodes, the number of job classes, the number of jobs per class, the scheduling policy of each node, and the transition probability of the network. Table II summarizes the levels of each parameter that is used to generate a queueing network. The routing matrix is generated using three methods: circular routing, circular with self-loops (referred here as looped routing), and fully random routing, as shown in Figure 2. In the circular routing , the nodes are connected sequentially to form a closed chain. The extension in circular routing adds self-loops at each node. The probability of the jobs going to the next station is generated from a uniform distribution between 0 and 1. In the random link structure, routing probabilities are randomly generated from a uniform distribution between 0 and 1.

Each network also has an infinite server node, referred to as the *Delay* throughout the study. If the *Delay* is not added to the network, queueing stations systematically operate in heavier loads and thus state space explosion may occur in the CTMC

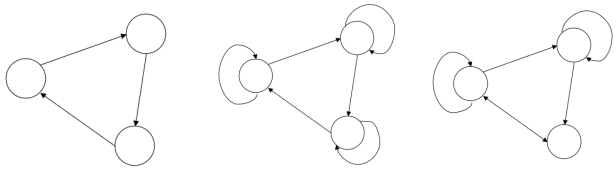


Fig. 2. Circular, looped, and random routing topologies used in models with $M = 3$ stations

TABLE II
SELECTION OF EACH FEATURE

Parameters	*Values*
Station scheduling	FCFS, PS, LCFS, SIRO, HOL
Number of stations	2, 3, 4
Number of classes	1, 2, 3
Service rates	Uniform in [1,3]
Number of servers	1, 2, 3
Number of per-class jobs	3, 4, 5, 6, 7

in the presence of multiclass FCFS and HOL nodes, making it more difficult to carry out comparisons with the exact model. All jobs are initialized at the *Delay* station.

Overall, 25000 queueing network models are generated and simulated for 5000 steps each[1]. A step here indicates that the state of the network has changed from state i to state j, whereby state i and state j are directly connected states in the sense that they differ only for a single job having changed its station. Therefore, each network has 5000 states.

The input of the model is a sequence of aggregated states as shown in (1). Each state is concatenated with the standardized time

$$\hat{t}_i = \frac{t_i - t_{min}}{t_{max} - t_{min}}$$

Therefore, the input matrix can be written as follow

$$x = \begin{bmatrix} \hat{t}_1 & n_{11}^{(1)} & n_{12}^{(1)} & \dots & n_{1C}^{(1)} & \dots & n_{NC}^{(1)} \\ \hat{t}_2 & n_{11}^{(2)} & n_{12}^{(2)} & \dots & n_{1C}^{(2)} & \dots & n_{NC}^{(2)} \\ \vdots & \vdots & \vdots & \dots & \vdots & \dots & \vdots \\ \hat{t}_m & n_{11}^{(m)} & n_{12}^{(m)} & \dots & n_{1C}^{(m)} & \dots & n_{NC}^{(m)} \end{bmatrix}$$

where $n_{ij}^{(t)}$ is the number of jobs of class j at station i at time t.

B. Evaluation setting

Three measurements are used to evaluate the performance of the model, which are accuracy, macro-F1 score and weighted-F1 score [22]. Accuracy is commonly used to represent the performance of a classifier. However, if the test labels are imbalanced, the accuracy cannot provide a valid result to judge the performance of the model. The F1 score is widely used to solve this problem.

The macro-F1 score computes the mean of the F1-scores of all classes, but in our setting, *Delay* station and "empty" class are easy to classify and exceed the number of other classes, and we do not care as much about them, so we lower their contribution to the final result in comparison to the others. As a result, the weighted-F1 score is calculated, with each class's weight factored into the final F1 score. The weight for *Delay* and empty class are set to $\frac{1}{12}$ and the weight for the remaining classes are set to $\frac{1}{6}$.

[1]The dataset is available at https://zenodo.org/record/5550315#.YVxf75pBxPY

TABLE III
THE PERFORMANCE OF RNN AND TRANSFORMER WITH FIRST 50, 500 AND 1000 STEPS AS INPUT DATA.

Model	*Sequence length*	*Accuracy*	*Macro-F1*	*Weighted-F1*
Baseline (SVM)	50	0.6736	0.5252	0.4499
	500	0.7332	0.6164	0.5572
	1000	0.7426	0.6328	0.5785
RNN-mean	50	0.7524	0.7371	0.6973
	500	0.6504	0.6244	0.5715
	1000	0.6458	0.6252	0.5831
RNN-last	50	0.4066	0.2937	0.1975
	500	0.4043	NaN	NaN
	1000	0.4488	NaN	NaN
Transformer	50	0.8104	0.7975	0.7654
	500	0.9018	0.8937	0.8767
	1000	0.9042	0.8968	0.8801

C. Model selection

In sections III and IV, we have presented three types of models that may be used for the problem under study: the baseline model SVM, the RNN model with LSTM cells and the transformer model. In this section, we compare the performance of these models with different length of continuous observations and sampled observations, then pick the best one to use in the rest of our experiment.

The RNN model is made up of ten layers of LSTM cell with a hidden dimension of 128. We consider two compression methods which are taking the mean of all outputs or only considering the final output. After checking the performance of transformer models with different structure, we used 3 encoders and 3 decoders to construct a transformer model and the hidden dimension is also set to 128 with 8 heads in the multi-head attention layers. A zeros matrix with the shape $(4, B, 128)$ is used as the input of the decoder.

1) Continuous observations: To check the effect of the sequence length, we try three different sequence length as the input to train the models. At first, only the first 50 time steps are considered as input data, followed by the first 500 and 1000 steps. The models are trained with 200 epochs and the batch-size and learning rate are set to 32 and 0.0001, respectively. The results are shown in Table III

From the results, we find that the RNN model which only takes the last state output as the final output learning nothing from the data. Also the mean F1 score and weighted-F1 score are NaN when the sequence length is set to 500 or 1000 states. The reason is the model can only classify the stations to LCFS and HOL queue which means the precision and recall on PS, FCFS and SIRO are 0, therefore, we cannot calculate the F1 scores for these three types of queue. Thus, the RNN model with last state compression is a invalid model in this problem.

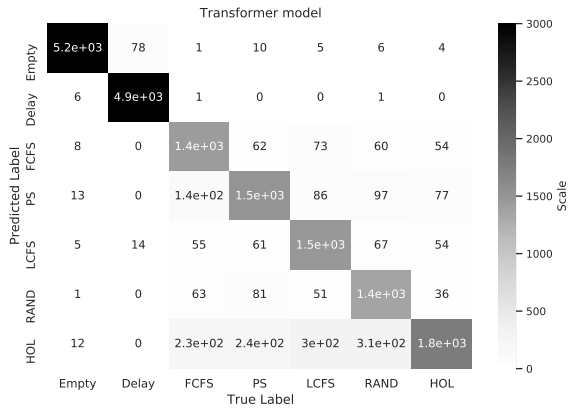


Fig. 3. The confusion matrix of transformer model trained by 1000 continuous states

The performance of the RNN model with mean compression decreases as the sequence length increases. Again, we find that the mean compression becomes less useful when the length of the sequence is too long in the RNN model, because it looses too much information by taking the mean. The reason for the mean compression method outperforms the last state compression is that the model considers the output of all state, which means that the error might propagate not only backwards in time but also directly to every state, However, it can only outperform the baseline SVM model with a sequence length of 50.

The transformer can produce the best performance among the three models. It performs much better in long sequence data with 90.42% of accuracy, 0.8968 of the macro-F1 score and 0.8801 of the weighted-F1 score compared to the one trained with the sequence length of 50 which accuracy, macro-F1 score and weighted-F1 score are 81.04%, 0.7975 and 0.7654 respectively. Furthermore, from Figure 3, the transformer model can clearly classify every type of scheduling policy.

2) Sampled observations: Next, we consider the situation that the state data is no longer continuous, but we observe the state periodically. Hundreds of thousands of requests are pushed into a web service system, thus tracking every network change is costly. Instead of memorizing all the states, the sensor can record the states periodically; for example, the sensor will save the state if the state of the network has changed every 5 times or 10 times. Another approach is to record the state of the network randomly. Both of these two approaches can possibly be used in a real system with a large number of requests. In this experiment, the RNN model with mean compression and transformer will be trained and tested with 4 types of data which are getting the states for every 5 steps (per5) or 10 steps (per10) and randomly taking 500 (rand500) or 1000 (rand1000) states from the 5000 states. The training parameters are identical to those used in the prior experiment. The results are shown in Table IV.

TABLE IV
THE PERFORMANCE OF RNN AND TRANSFORMER WITH PERIODIC AND RANDOM DATA.

Model	*Dataset*	*Accuracy*	*Macro-F1*	*Weighted-F1*
Baseline	per5	0.7566	0.6598	0.6144
	per10	0.7631	0.6617	0.6107
	rand500	0.5352	0.3668	0.2959
	rand1000	0.5596	0.3940	0.3221
RNN-mean	per5	0.4541	NaN	NaN
	per10	0.6585	0.6247	0.5628
	rand500	0.4019	NaN	NaN
	rand1000	0.4623	0.4007	0.3141
Transformer	period 5	0.8959	0.8882	0.8698
	per10	0.8958	0.8900	0.8718
	rand500	0.8359	0.8242	0.7951
	rand1000	0.8496	0.8408	0.8145

TABLE V
THE PERFORMANCE OF TRANSFORMER TRAINED WITH PERIODIC AND RANDOM DATA AND TESTED ON THE OTHER TYPES OF DATA.

Train type	*Test type*	*Accuracy*	*Macro-F1*	*Weighted-F1*
per5	rand500	0.6012	0.5642	0.4933
	rand1000	0.7256	0.7022	0.6531
rand1000	per5	0.8575	0.8497	0.8249
	per10	0.7561	0.7374	0.6942

Transformers trained using periodic data perform better than those trained with data taken at random from 5000 states. Although the models trained with the periodic sequences seem to have higher evaluation scores, we think that the models trained with the random sequence can also provide valuable results on the periodic data because the periodic data is the subset of the random selection. To prove this, the models trained with periodic data are tested on the random sequence. Meanwhile, the models trained with the random sequence are used to classify the scheduling policy on the periodic data.

From Table V, the model trained with the random 1000 states data can also perform well on data record the state every 5 steps with 85.75% of accuracy, 0.8497 of the macro-F1 score and 0.8249 of the weighted-F1 score. On the contrary, the model trained with periodic data cannot give a satisfying classification on the randomly selected data. Therefore, the transformer trained with randomly selected data is more robust than the one trained with periodic data. In the following experiments, the models are trained with the random selected setting. We notice that the performance of the model is also affected by the sequence length. We test the transformers which are trained by the 500 and 1000 randomly selected states data on the test data with different sequence length from 50 to 1600.

As shown in Figure 4, the accuracy and F1 scores reach their maximum when the sequence length of the test data is the same as the training data. When the sequence length of the test data increases, the performance drops marginally, but the model can still give reasonably good classification. However, if the sequence length of the test data decreases, the accuracy and F1 scores drop dramatically. Therefore, we believe the model can handle sequences longer than the training sequence, but it fails to give an useful classification for sequences shorter

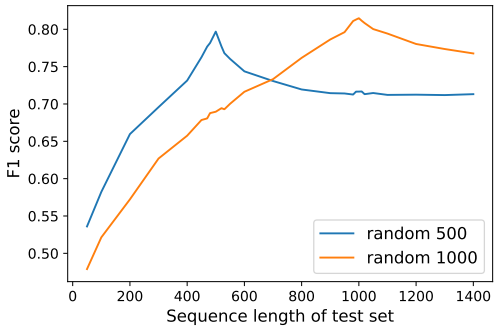


Fig. 4. The weighted-F1-score of transformers on test data with different sequence length

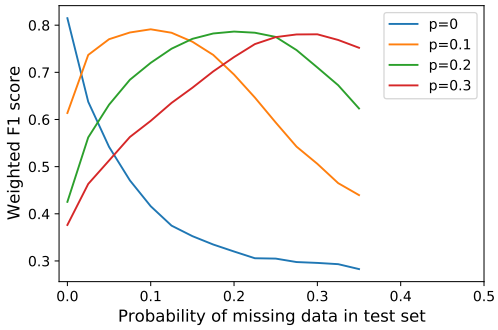


Fig. 5. The weighted-F1 score of the transformer with missing data. (The accuracy and macro-F1 score have nearly identical trend)

than the training sequence. So there is a trade-off between the length of the sequence and the performance of the model because the model trained on short sequences can work in a broader area but with lower accuracy.

D. Training with missing data

In this experiment, we study the robustness of the transformer model by training the model with missing values. There are two different types of missing values. The first type is that the number of jobs n_{ij} is missing with some probability, which means the sensor recording the state of the network is failed by chance. To simulate this situation, we can set the number of jobs at each station to -1 with probability p to represent the missing values. The other type is called missing features, which means the sensor cannot monitor some classes of jobs at some stations. In this type of missing value, the columns of the missing features are set to -1, and, for each training sample, the missing features are randomly chosen. In the next experiment, we call the data set no masked data if it does not contain a missing value.

1) Missing values: For the first type of missing values, each value n_{ij} is missing with probability p, which equals to 0, 0.1, 0.2 and 0.3. The data set is masked with 4 different values of p and used to train the model separately. The training data are selected randomly with 1000 states, just like we did in Section V-C2.

From Figure 5, we can find that the models give the best performance when the test data are masked by the corresponding probability. If the probability of missing data in the test set is different from the training set, the accuracy and the F1 score of the model will decrease. We can also discover that the model trained with the complete data (blue curve) significantly decreases when the probability of missing value increases, which means this model only provides valid classification if the test data does not contain the missing value. The reason behind this is that the training set does not contain -1, which represents the missing value. Therefore, if the test set has -1, the model cannot process the information of missing value.

On the other hand, the models trained with missing data are more robust than the model trained with complete data.

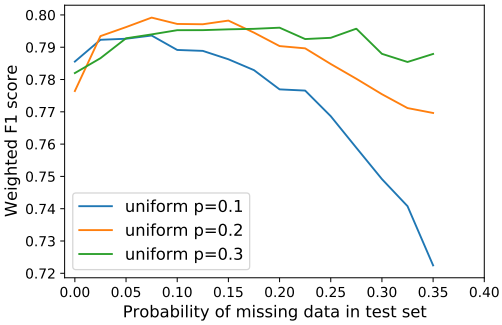


Fig. 6. The weighted-F1-score of transformer with missing data if the probability p is chosen from an uniform distribution

For example, the model trained with 10% of missing data (orange curve) gives a useful classification when the test data are masked with the probability between 0.025 to 0.175 where the accuracy and the mean F1 score are around 0.8 and the weighted-F1 score is greater than 0.7. If the probability of missing value is outside this interval, the performances drop dramatically. The models which are trained with 20% and 30% of missing data have a similar property. The result of the models can only perform well in a specific interval of the missing probability that can be due to the fact that the distribution of missing value on the test data is different from the training data. The models do not have knowledge on other distribution of missing value.

To prove this, we choose the probability of missing value in a sample from an uniform distribution with low bound equal to 0 and upper bound equal to $2p$. Therefore, we have more uncertainty on the missing value and get the result as shown in Figure 6. The interval where the model can give promising classification becomes larger, and we can conclude that the model can provide good performance if the testing data stay in similar distribution of the training data.

2) Missing features: In our experiment, the maximum number of stations in a queueing network is 4, and the maximum number of class of jobs is 3, so the total number of features in a state vector is 12 plus the time. For the second type of missing

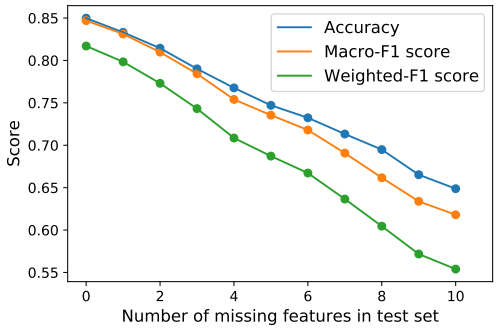


Fig. 7. The performance of transformer with missing features

values, the number of missing features is set between 0 and 10 except the time, which means the time will never miss. More precisely, if we select m missing features, all the training data are miss m features. Unlike the first type of missing values, we can easily observe how many features are missing in the test data. Therefore, we can select the right trained model according to the test data. So the models are evaluated by the test data with the corresponding number of missing features. The data selection and training settings are the same as the previous experiment.

Figure 7 shows the accuracy, mean F1 score and weighted-F1 score of the model trained by the corresponding number of missing features and tested by the test data set. All three curves have a similar trend and the scores decrease linearly when the number of missing features is increased. Models that were trained with 0 to 6 missing features can nevertheless perform reasonably well. We think that the missing features are randomly selected so the padding features can be selected and there will not be a significant effect on the model if these features are missing. When the number of missing features is greater than 7, the mean F1 score and weighted-F1 score drop faster than the accuracy. The reason is that the models cannot give a precise classification on FCFS, LCFS, SIRO, PS and HOL queue, but they can still classify the *Delay* and empty stations. This is the case considering the fact that every sample data must have a *Delay* station and the total number of empty stations is much larger than the other types of queue.

VI. Conclusion

This paper focuses on a novel problem of estimating the scheduling policies of a queueing network from aggregate sample path traces. The transformer model is shown to be an effective approach to solving the problem when an extra input is provided to the decoder in the transformer model. This input provides a set of learnable parameters for the decoder to learn a particular part of the key-value pairs passed on by the encoder. The transformer model has been successfully used to classify scheduling policies from a sequence of sampled, non-continuous, states of the queueing network with much greater accuracy than SVMs or RNNs.

Future work may explore the application of this work to actual system measurements and the challenges arising with extended queueing features not considered in this paper.

References

[1] G. Bolch, S. Greiner, K. S. Trivedi, H. de Meer, and H. de Meer, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. John Wiley & Sons, Incorporated, 2006.

[2] S. Spinner, G. Casale, F. Brosig, and S. Kounev, "Evaluating approaches to resource demand estimation," *Performance Evaluation*, vol. 92, pp. 51 – 71, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166531615000711

[3] W. Wang, G. Casale, A. Kattepur, and M. Nambiar, "Maximum likelihood estimation of closed queueing network demands from queue length data," *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pp. 3–14, 2016.

[4] W. Wang, G. Casale, and C. Sutton, "A bayesian approach to parameter inference in queueing networks," *ACM Transactions on Modeling and Computer Simulation*, vol. 27, no. 2, 2016.

[5] F. Baskett, K. M. Chandy, R. Muntz, and F. Palacios, "Open, closed and mixed networks of queues with different classes of customers," *Journal of the ACM*, vol. 22, no. 2, pp. 248–260, 1975.

[6] G.Sh.Tsitsiashvili and M.A.Osipova, "Parameter estimation for product-form distributions of queueing networks," *Probl Inf Transm*, vol. 45, pp. 400–405, 2009.

[7] G. Garbi, E. Incerto, and M. Tribastone, "Learning Queuing Networks by Recurrent Neural Networks," *arXiv e-prints*, p. arXiv:2002.10788, Feb. 2020.

[8] T. G. Kurtz, "Solutions of Ordinary Differential Equations as Limits of Pure Jump Markov Processes," *Journal of Applied Probability*, vol. 7, pp. 49–58, 1970.

[9] M. Raeis, A. Tizghadam, and A. Leon-Garcia, "Predicting distributions of waiting times in customer service systems using mixture density networks," pp. 1–6, 2019.

[10] C. Sutton and M. I. Jordan, "Bayesian inference for queueing networks and modeling of internet services," *arXiv e-prints*, p. arXiv:1001.3355, Jan. 2010.

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[12] D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang, "Quality-of-service in cloud computing: modeling techniques and their applications," *J. Internet Serv. Appl*, vol. 5, no. 1, pp. 11:1–11:17, 2014.

[13] C. Cortes and V. Vapnik, "Support-Vector Networks," *Machine Learning*, vol. 20, pp. 73–297, 1995.

[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[15] Q. Wang, B. Li, T. Xiao, J. Zhu, C. Li, D. F. Wong, and L. S. Chao, "Learning Deep Transformer Models for Machine Translation," *arXiv e-prints*, p. arXiv:1906.01787, Jun. 2019.

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv e-prints*, p. arXiv:1810.04805, Oct. 2018.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv e-prints*, p. arXiv:1512.03385, Dec. 2015.

[18] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *3rd International Conference for Learning Representations 2015*, 2014.

[20] G. Casale, "Integrated performance evaluation of extended queueing network models with line," in *Proceedings of the Winter Simulation Conference*, ser. WSC '20. IEEE Press, 2020, p. 2377–2388.

[21] J. F. Pérez and G. Casale, "Line: Evaluating software applications in unreliable environments," *IEEE Transactions on Reliability*, vol. 66, no. 3, pp. 837–853, 2017.

[22] N. Chinchor, "Muc-4 evaluation metrics," *Proceedings of the 4th conference on Message understanding*, pp. 22–29, 1992.