Deep Learning Models for Automated Identification of Scheduling Policies

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

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

Abstract

Queueing network models are commonly used 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 affect in general the stationary distribution of the model, 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. To address the shortage of methods for this parameter identification problem, we propose a deep learning technique based on transformer models, a popular technique in natural language processing. Our approach can classify the scheduling policy of the stations in a queueing network from a sample path of the joint network state, or from an aggregate thereof. We show that the transformer model delivers high-classification precision and recall, improving significantly over support vector machines or simpler recurrent neural networks.

CCS Concepts: • **Networks** → **Network performance modeling**; • **Computing methodologies** → **Artificial intelligence**.

Keywords: queueing network, scheduling policy, neural networks

ACM Reference Format:
Yichong Chen and Giuliano Casale. 2018. Deep Learning Models for Automated Identification of Scheduling Policies. In *ICPE '21: ACM/SPEC Int'l Conference on Performance Engineering, June 03–05, 2018, ICPE, NY*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

ICPE '21, June 03–05, 2018, ICPE, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 Introduction

Queueing network models are commonly used stochastic models to represent congestion in software applications and service-based systems [4]. A natural question that arises in connection to the recent growth of AI technologies is the identification of AI and machine learning methods that can be help to automatically identify a model from empirical data.

Over the last decade, a comprehensive range of studies have investigated estimating service demands, which are important drivers of model prediction accuracy [20] and one of the main sub-problems in the performance model identification field. For example, in [24], the authors apply maximum likelihood estimated to predict demands in closed queueing networks. Similarly, the work in [25] tackles the same problem via Gibbs sampling and Monte Carlo integration [3]. Prior art in parameter estimation also includes open models, such as [11] which estimates service and arrival rates limiting distribution based on the number of jobs in the node. AI and machine learning are also used in some studies. For example, in [10], the authors apply a fluid approximation to the queueing network [17] 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 [19] 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 important to identify with high confidence the scheduling policy in use within a particular system. For example, [21] proposes different algorithms, depending on the scheduling policy, to estimate the missing data within a finite sample path of a queueing network. 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.

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

Motivated by the above considerations, this paper proposes a transformer model [22] to automatically identify the scheduling policies of a network of resources, modeled as queueing stations, assuming that partial measurements are obtained of their state. To the best of our knowledge, the present work is the first one to develop an AI method to learn scheduling characteristics from partial monitoring data and accounting for the presence of queueing phenomena. We show in particular that transformer models, commonly used in natural language translation problems, can effectively be adapted to the queueing model identification problem we study. Experiments using simulation data confirm that it is possible to recover with relatively high accuracy the scheduling characteristics of the system that generated the data using a supervised learning approach. The proposed method can result in both high accuracy and F1-score, both exceeding 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 classes of AI models applicable to the problem at hand. Section 4 introduces the proposed transformer model for queueing model identification. Section 5 gives experimental results , followed by conclusions in Section 6.

2 Background

Queueing network models have been extensively studied in the scientific literature and applied to many computer and communication systems [1]. These models are composed by a network of M queueing stations, at which jobs receive service. Service times have a random duration, according to some given statistical distribution. Jobs may either visit the system and leave (open classes), or perpetually cycle in the network (closed classes). Routing is probabilistic. At each station, the 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) [4]. 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, within that group, they are served in arrival order.

Table 1. State representation for a given station

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

Let us consider a model with M stations and where jobs can belong to one of R service classes. No class switching is allowed. We assume service times to be exponentially distributed.

When we represent such systems as a continuous-time Markov chain, the state vector for a given station can be represented as in Table 1, 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.
- m: The number of servers in the station

It would be fairly simply to classify scheduling by observing the detailed evolution of the states shown in Table 1. The problem we consider here is however to do so by observing aggregate state vectors

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

where $x_{ij}(t)$ is the number of jobs of class j at station i and symbol t denotes the time of the observation. The sample path of aggregate states has a more compact representation than using the exact state representation. 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, because it does count only their aggregate number. However, this simplification also implies that two models with different scheduling policies may produce the same sample paths, in terms of aggregate variables, motivating our investigation into learning methods to resolve the ambiguity that this produces. 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 time series $x(t)$, $t \in T$.

3 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 for example in a web server, it is not uncommon to see hundreds of thousands of requests received within seconds, making the exact tracking of the system state fairly prohibitive. 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 more than one queueing station to model its performance. Therefore, collecting all the states at every node in a reliable fashion becomes more difficult. Moreover, some data may be missing, further complicating the learning process.

Secondly, in the presence of aggregation, it is not easy to identify from the sample paths clear features to distinguish the policies. Figure 1 illustrates this by showing the state sequence at a single-server queueing station under different scheduling policies. HOL scheduling, not shown in the figure, is qualitatively similar. We can see that, from the figures, it is difficult to recognize immediate patterns to classify the scheduling policy from the traces. Confronted with this challenge, we propose to use learning methods to solve the model identificaiton problem.

In this section, we introduce three possible architectures that may be used for this problem, leveraging SVMs, RNNs, and transformers. As our experimental validation later shows that the transformer vastly exceeds the performance of the other two techniques, thus we further expand in Section 4 the discussion of this method which is only here briefly introduced.

3.1 Support Vector Machines

The support vector machine (SVM) is a traditional machine learning model introduced by Corinna Cortes and Vladimir Vapnik in 1995 [7]. SVMs are binary linear classifiers that find a hyper-plane which maximise the gap between two classes and assigns the new samples to each category with the hyperplane.

The SVM draws a linear decision boundary to separate two classes, but the model will fail if they are not linearly separable. The input data can then be projected to a higher dimension using a so-called kernel function. The kernel function, if properly chosen, can make the data linearly separable in the higher dimensional space.

The radial basis function (RBF) is commonly used in classification problems and also used throughout the present work. We notice that the SVM can only separate two classes, so for multi-class classification, the classifier circumvents this limitation by considering one-versus-one or one-versus-all sub-problems and combining the results in a multi-class classifier.

SVMs can be applied to time series data as follows. We consider the states of a station within a period as the 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.

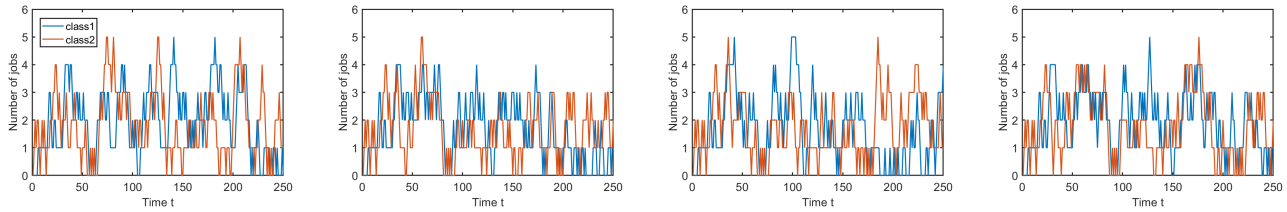
3.2 Recurrent Neural Networks

In deep learning (DL) and natural language processing (NLP), recurrent neural networks (RNN) are often used to learn patterns within a time series.

However, simple RNNs are prone to the vanishing gradient problem and related numerical issues. For example, since the loss is back-propagated through the entire sequence, if the gradients are small, then multiplying a long series of small numbers will push the gradient towards zero, and the parameter update will also be prone to fail. Moreover, when the gradients become excessively large, the gradient descent method may also fail due to numerical issues. These problems may be addressed by replacing the naive RNN cell with a long-short term memory (LSTM) cell [14].

As shown in Figure 2, the model can have more than one layer of LSTM cells. In a model with multiple LSTM layers, the next LSTM layer takes the output from the previous layer as its input. Every input state will produce an output, which means that the number of outputs in the final layer equals the sequence length n given in input to the model. If we indicate the number of traces used in a training batch with B and the output dimension of the LSTM cells with H, also called hidden dimension, then the shape of the training output after the LSTM cells is (B, n, H) where the elements in the bracket represent the dimension in the corresponding tensor representation. Next, all the outputs from a trace need to be reduced into a single vector to produce a classification decision. This may either be done by averaging all outputs, or only by considering the output associated to the last state, so that the shape of the output matrix after compression has shape (B, H). A linear layer then receives the compressed vectors and convert them to the required size.

The above model may be generalized so that if the number of stations is different in different queueing networks used for training or testing, then we are still able to classify scheduling using a single model that only has a bound on the maximum number of stations that the network can include. Let us define the maximum



(a) FCFS **(b)** LCFS **(c)** PS **(d)** SIRO

Figure 1. Examples of sample paths for two job classes under different scheduling policies. Due to job count aggregation, the scheduling identification problem is non-trivial.

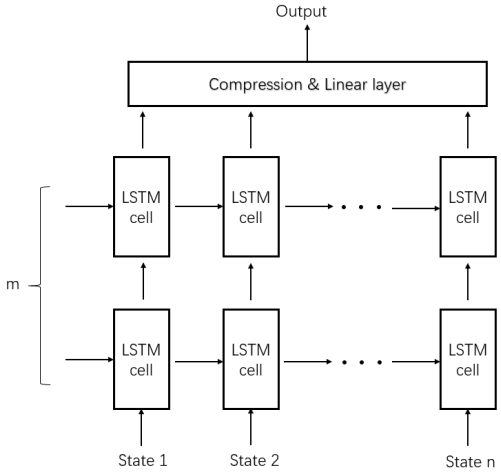


Figure 2. The structure of the LSTM model.

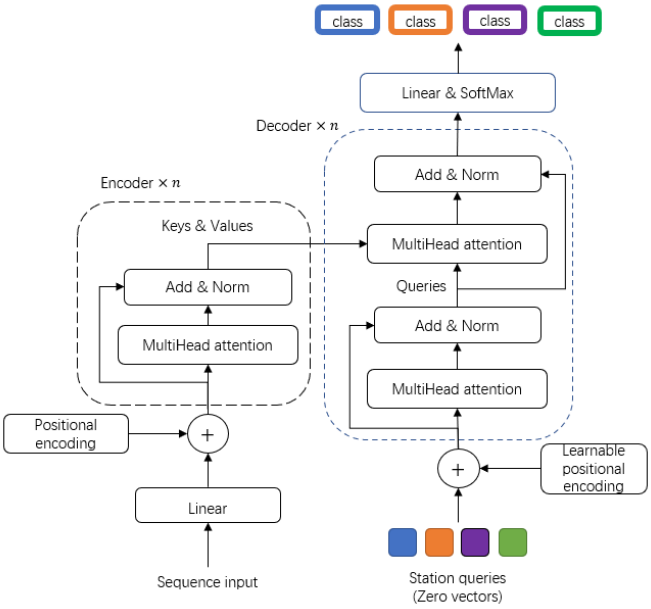


Figure 3. Proposed Transformer architecture

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.

3.3 Transformer model

The transformer model [22] has made outstanding achievements on solving NLP problems such as machine translation [23] and constructing a language model [8]. In [2], the paper shows that a new method named attention has a better performance than simple compression methods. To apply this model to the problem at hand, we introduce a modified transformer architecture as shown in Figure 3; details of this architecture are described later in the next section. At a high-level, the model takes the state sequence as its input and then outputs the probability of a station using a particular type of scheduling policy. In the standard transformer, the model gives predictions sequentially in machine translation, but in our case, we predict the scheduling policy of n_{query} stations simultaneously. The model assumes a maximum number of stations but is applicable to networks of different sizes. If fewer stations are contained in the network than maximum, the model will classify the scheduling policy of the unused stations in a class that we label "empty". In practical use, we set $M \leq n_{query}$ so that it is large enough to contain all the stations M for every possible model considered during training and testing. That is, the transformer may be trained or tested with queueing networks having any number of stations M, provided that $M \leq n_{query}$. In the next sections, we describe in finer details the transformer architecture and its training process.

4 Transformer Model Architecture

4.1 Overview

In this section, we describe in details the architecture shown in Figure 3. The meaning of the blocks in the model architecture is as follows. The transformer model is a encoder-decoder model. The encoder maps the state

sequence to a new sequence of representations which 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 in Figure 3. Therein, the position encoding blocks embed the order of the sequence and the multi-head attention blocks process the correlation between each states within a sequence. At last, the data is processed with a linear feed-forward network and the softmax function, so that the output of the model represent the probability of a particular scheduling policy being in use at each station.

4.2 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 [22] to the queueing domain. This mechanism can process the relation between any two states in the sequence, and its output is a vector $\boldsymbol{a} \in \mathbb{R}^{n\times d_v}$, 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 the 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

$$a_i = \text{softmax}\left(\frac{\boldsymbol{q}_i \boldsymbol{k}_j^T}{\sqrt{d_k}}\right)\boldsymbol{v}_j \quad i = 1, \ldots, n \tag{1}$$

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

Here, the dot product is applied to every query with all the keys and 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, we use more than one attention function in a layer. Each attention function is named *attention head* and focuses on different features of the sequence. These attention heads form the multi-head attention layer which is written as follows [22],

$$\text{MH}(Q, K, V) = \text{Concat}(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_h)W^O \tag{3}$$

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

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. More precisely, if we take the attention outputs $\boldsymbol{a}_i \in \mathbb{R}^{n\times d_v}$, 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.

4.3 Transformer Encoder

Suppose the input state sequence is $\boldsymbol{x} \in \mathbb{R}^{n\times d}$ 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 $\boldsymbol{x}' \in \mathbb{R}^{n\times d_{model}}$. In our experiment, adding this FFN can significantly improve the performance of the model.

Unlike the RNN model which inherently take the order of sequence into account, the encoder and decoder of the transformer model only contain the multi-head attention and the FFN, and both of these networks do not consider the order of the sequence. Therefore, a position encoding P is added to the input to embed the order of the sequence. Although the order can be represent as the number in the binary format, it is space consuming and depends on the assumption of the maximum length of the sequence. Therefore, in [22], the sine and cosine functions are used as the position 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 do not depends on the values of the input but on d_{model}. As shown in Fig 4, the row vectors in the positional encoding matrix are different from each other, so 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 $\boldsymbol{x}'$), 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 [12].

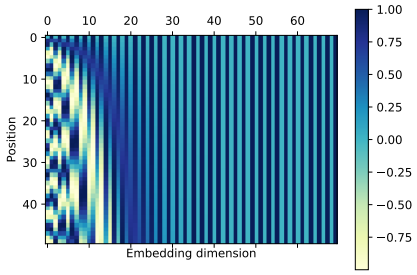


Figure 4. The position encoding matrix with $d_{model} = 70$ and $T = 50$

We finally take the batch normalization on the output in order to accelerate the training and reduce the covariate shift [15]. In our model, there are more than one encoder. The output of the previous encoder becomes the input of the next encoder until the last one.

4.4 Transformer Decoder

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. For example, the translation model produces the words one by one, so that the output of the previous translation becomes one of the inputs of the decoder. However, in our model, we classify the scheduling policy of all the stations simultaneously. Thus, the model does not have the output of the previous prediction. 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. Therefore, a learnt position encoding is applied and, similar to the position encoding in the encoder, the input is added with the position encoding.

In the decoder, the input is first processed with a multi-head attention network analogous to the one in the encoder. Then another multi-head attention is applied, but this layer takes the output of the previous attention network as the query. The output of the encoder is then processed as the key and value to this attention layer. Therefore, the second attention network considers both the input of the decoder and the output of the encoder.

4.5 Prediction FFN and loss function

Finally, the prediction is computed with an FFN with softmax as the activation function. The output of the transformer with shape (n_{query}, d_{model}) is projected to a matrix with shape (n_{query}, C).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 (Eq.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 want the model can also obtain good performance on the other policies. So the loss of each class needed to be weighted. The weight of the *Delay* and empty class should be much small than the others. So the loss function is

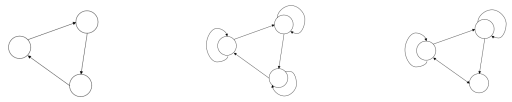
$$\text{Loss}(\text{output}, y) = -\sum_{i=1}^{N}\sum_{j=1}^{C} \text{Weight}(y_{ij})(y_{ij}\log(\text{output}_{ij})) \tag{7}$$

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 minimise the cross-entropy in order to maximise the probability of giving the correct classification. This can be done by updating the parameters with a gradient descent algorithm. There are many useful gradient descent algorithm such as Adagrad [9] and RMSprop [13]. In our experiments, we use Adam [16].

5 Experiment

5.1 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 [5, 18]. 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 2 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 5. In the circular routing in Figure 5a, the nodes are connected sequentially to form a closed chain. The extension in Figure 5b 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. To generate in this scenario also sparse routing matrices, transition probabilities smaller than 0.05 are rounded below to 0.



(a) circular routing **(b)** looped **(c)** random

Figure 5. Network topologies used in models with $M = 3$ stations

Table 2. 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

An infinite server node, labelled in the rest of the paper as the *Delay*, is also included in each network. 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 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. 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 following matrix represents a sample input

$$\boldsymbol{x} = \begin{bmatrix} t_1 & n_{11}^{(1)} & n_{12}^{(1)} & \dots & n_{1C}^{(1)} & \dots & n_{NC}^{(1)} \\ t_2 & n_{11}^{(2)} & n_{12}^{(2)} & \dots & n_{1C}^{(2)} & \dots & n_{NC}^{(2)} \\ \vdots & \vdots & \vdots & \dots & \vdots & \dots & \vdots \\ t_{5000} & n_{11}^{(5000)} & n_{12}^{(5000)} & \dots & n_{1C}^{(5000)} & \dots & n_{NC}^{(5000)} \end{bmatrix}$$

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

5.2 Evaluation setting

The confusion matrix is commonly used to analyse the performance of a classifier by calculating the measurements from the matrix. Three measurements are used to evaluate the performance of the model, which are accuracy, macro-F1 score and weighted-F1 score [6]. 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. To overcome this problem, the F1 score is commonly used. To adjust the multi-class confusion matrix to the binary confusion matrix, we can rewrite the matrix to one versus all form for every class. In this way, the F1 score for each class can be calculated and weighted to obtain the F1 score for the classifier. Two weighting methods are considered in our experiment, which are the macro-F1 score and the weighted-F1 score.

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 compared to the others. This leads to the weighted-F1 score so that every class has its weight in calculating 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 $\frac{1}{6}$.

5.3 Model selection

In sections 3 and 4, we have presented three types of neural network 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 sequence data and not continuous data and select the best one and apply it in the rest of our experiment.

The RNN model contains 10 layers of LSTM cell and set hidden dimension to 128. We consider two compression methods the RNN which are taking the mean of all outputs or only considering the final output. 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.

5.3.1 Continuous observations.

To check the effect of the sequence length, we try three different sequence length as the input to train the models. We only consider the first 50 time steps as the input data at the start, then the first 500 steps 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. Adam is applied as the optimiser on both models. The results are shown in Table 3

From the results, we find that the RNN model which only takes the last state output as the final output does not learn anything 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. From Figure 6, the model trained with sequence length of 1000 can classify

Table 3. 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	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

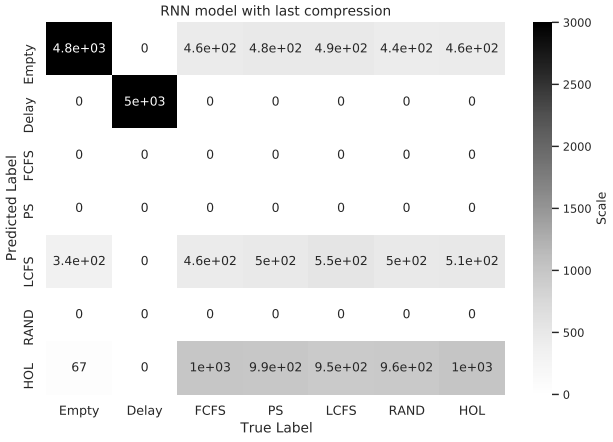


Figure 6. The confusion matrix of RNN model with last state compression trained by 1000 continuous states

the empty and *Delay* class, but it 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.

The RNN model with mean compression performance has a better performance with 75% accuracy, and the macro-F1 score and weighted-F1 score are 0.7371 and 0.6973, respectively when the sequence length of the data is 50. When the sequence length increase, the performance of this model decrease. Again, we find that the mean compression becomes less useful when the length of the sequence is too long in the RNN model, because by taking the mean, it looses too much information. The reason for the mean compression method outperforms

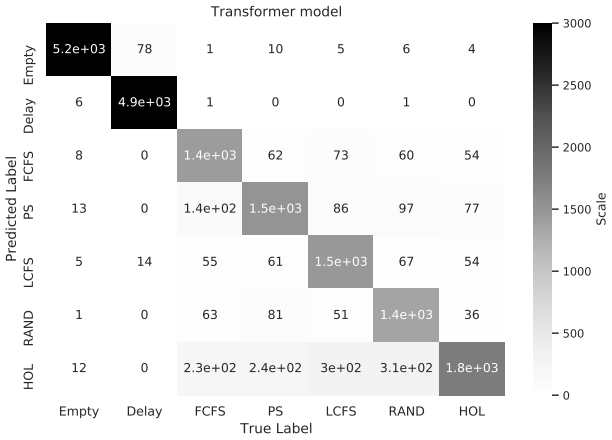


Figure 7. The confusion matrix of transformer model trained by 1000 continuous states

the last state compression is that the model considers the output of all state, which means the error can not only back-propagate through time but also directly to every state, which is similar to the residual connection. Therefore, the gradient vanish and gradient explode are less likely to happen compared to the last state compression which the error can only back-propagate through time. However, it can only outperform the baseline SVM model with a sequence length of 50.

The transformer can produce the best performance among 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 compare to the one trained with the sequence length of 50 which accuracy, macro-F1 score and weighted-F1 score are 8104%, 0.7975 and 0.7654 respectively. Furthermore, from Figure 7, the transformer model can clearly classify every type of scheduling policy.

5.3.2 Sampled observations. Next, we consider the situation that the state data is not continuous anymore, but we observe the state periodically. In a web service system, hundreds of thousands of requests are moved into the system, so recording every change of the network is expensive. 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

Table 4. 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

(per10) and randomly taking 500 (rand500) or 1000 (rand1000) states from the 5000 states. The training settings are the same as the previous experiment. The results are shown in Table 4.

Unlike the continuous sequence experiment, the RNN model with mean compression struggle to learn the information from the discrete sequence data, though the model performs better than random guess when we can record the state every 10 steps (500 steps). On the contrary, the transformers still give satisfying results on both periodic and random sequence data. An interesting result occurs on the baseline model: the model gives over 75% accuracy if it is trained on periodic data, but when the model is applied on the random sequence data, it fails to give the correct classification. After checking the test data, we find that some test data are the same as the training data. Therefore, the periodic data will be the same. However, if we consider the random sequence, the test data are different from the training data even if they come from the same queueing network. Thus, using the random sequence data will be a more suitable choice to prove the effectiveness of the transformer model.

The transformers trained with periodic data have a better performance than those trained with the data, randomly chosen 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. In the meantime, the models trained with the random sequence are used to classify the scheduling policy on the periodic data.

Table 5. 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

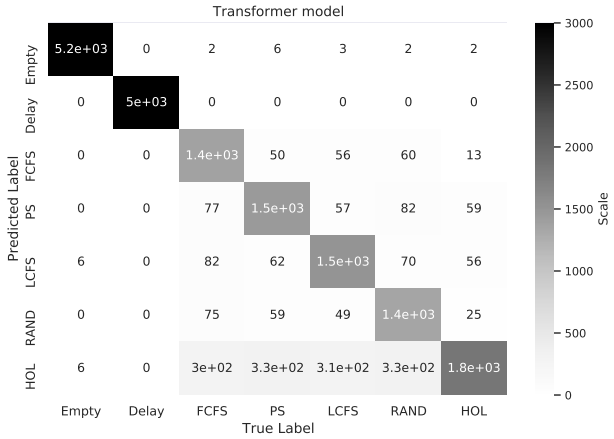


Figure 8. The confusion matrix of transformer model trained by 1000 random selected states

From Table 5, 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 9, 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 performances slightly decreases, and 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 think that the

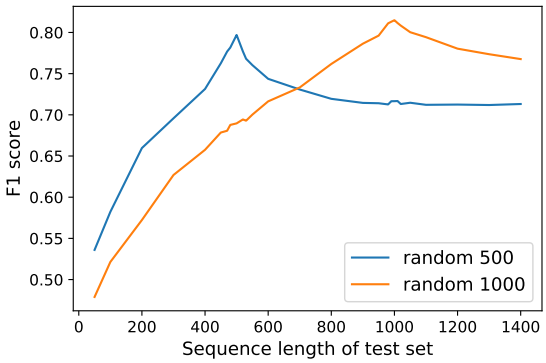


Figure 9. The weighted-F1-score of transformers on test data with different sequence length

Table 6. The performance of transformers with different structure.

Structure	*Params*	*Accuracy*	*Macro-F1*	*Weighted-F1*
1 encoder 1 decoder	466567	0.7728	0.7550	0.7146
3 encoders 3 decoders	1392263	0.8496	0.8408	0.8145
6 encoders 6 decoders	2780807	0.8531	0.8422	0.8166

model can handle the sequence that is longer than the training sequence, but for the sequences shorter than the training sequence, the model fails to give a useful classification. 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.

5.3.3 Transformer structure selection. Next, we consider to find a suitable structure of the transformer that the model contains a reasonable amount of parameters and provides precise classification. This can be done by changing the number of encoder and decoder in the model.

As shown in Table 6, the transformer contains 3 encoders and 3 decoders performs better than the one with only 1 encoder and 1 decoder. However, the model with 6 encoders and 6 decoders does not have much improvement on classifying the scheduling policy compare to the 3 encoders and 3 decoders model. When training the largest model, we need to adjust the batch size to 16 in order to fit the storage of the GPU (Graphic Process Unit) and it takes over 37 hours to finish the training process. The medium model that takes 18 hours to train for 200 epochs is much cheaper to train than the largest model.

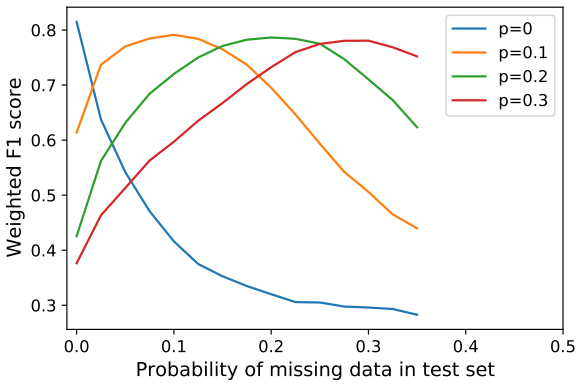


Figure 10. The weighted-F1 score of the transformer with missing data. (The accuracy and macro-F1 score have nearly identical trend)

In conclusion, the transformer with 3 encoders and 3 decoders is an appropriate structure in our problems.

5.4 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. If the data set does not contain a missing value in the following experiment, we name it the no masked data.

5.4.1 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 5.3.2. The models are trained with 200 epochs and learning rate and batch size are equal to 0.0001 and 32, respectively.

From Figure 10, 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 that the model trained with the complete data (blue curve) significantly decreases when the probability of missing value increases, which

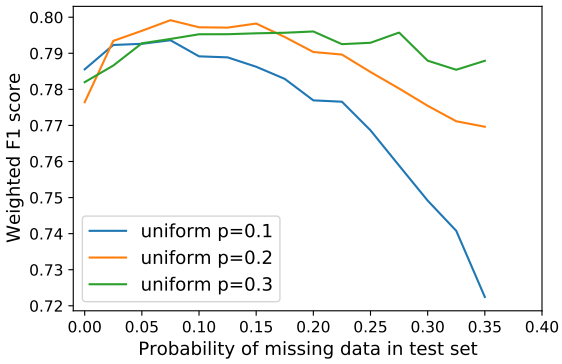


Figure 11. The weighted-F1-score of transformer with missing data if the probability p is chosen from an uniform distribution

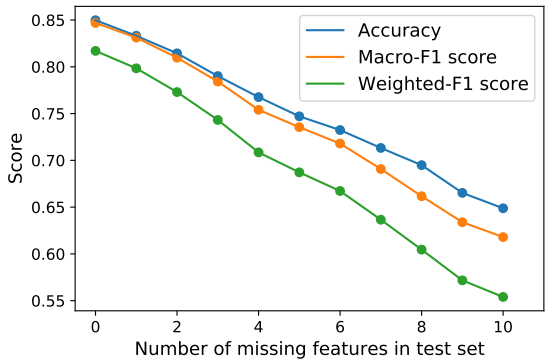


Figure 12. The performance of transformer with missing features

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. 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 11. 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.

5.4.2 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 values, the number of missing features is set between 0 and 10 except the time, which means the time will never miss. In more precise, if we select m missing features, all the training data are miss m features which means we train 11 models in this experiment. 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 12 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. Both three curves have a similar trend and the scores decrease linearly when the number of missing features is increased. The models which trained with 0 to 6 missing features can still provide reasonably good performance. 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.

6 Conclusion

This paper focuses on a novel problem of estimating the scheduling polict of a queueing network from aggregate sample path traces. The transformer model is shown to be an effective approach to solve the problem, provided

that an extra input is supplied to for 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 on this model to classify scheduling policies from a sequence of sampled, non-continuous, states of the queueing network with much greater accuracy than SVMs or RNNs.

A known limitation of the transform model is that the dimension of the features of the input data is fixed, but the dimension of the state vector is depended on the size of the queueing network. Therefore, we need to embed the state of the queueing network to a large-enough fixed dimension, so that the transformer model can work on large instances.

We plan to extend our work by considering other types of scheduling policy and networks and also experiment the model on a real system.

Acknowledgements

The authors wish to thanks Anthony J. Field for his constructive feedbacks on an earlier version of this work. The work of G. Casale has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825040.

References

[1] Danilo Ardagna, Giuliano Casale, Michele Ciavotta, Juan F. Pérez, and Weikun Wang. 2014. Quality-of-service in cloud computing: modeling techniques and their applications. *J. Internet Serv. Appl* 5, 1 (2014), 11:1–11:17. pages 2

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv e-prints*, Article arXiv:1409.0473 (Sept. 2014), arXiv:1409.0473 pages. arXiv:1409.0473 [cs.CL] pages 4

[3] F Baskett, K. Mani Chandy, R.R. Muntz, and F.G Palacios. 1975. Open, closed and mixed networks of queues with different classes of customers. *J. ACM* 22, 2 (1975), 248–260. pages 1

[4] Gunter Bolch, Stefan Greiner, Kishor S. Trivedi, Hermann de Meer, and Hermann de Meer. 2006. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. John Wiley & Sons, Incorporated. pages 1, 2

[5] Giuliano Casale. 2019. Automated Multi-paradigm Analysis of Extended and Layered Queueing Models with LINE. *Proc. of ACM/SPEC 2019, ACM Press* (2019). pages 6

[6] Nancy Chinchor. 1992. MUC-4 EVALUATION METRICS. *Proceedings of the 4th conference on Message understanding* (1992), 22–29. pages 7

[7] Corinna Cortes and Vladimir Vapnik. 1995. Support-Vector Networks. *Machine Learning* 20 (1995), 73–297. pages 3

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv e-prints*, Article arXiv:1810.04805 (Oct. 2018), arXiv:1810.04805 pages. arXiv:1810.04805 [cs.CL] pages 4

[9] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for misc Learning and Stochastic Optimization. *Journal of Machine Learning Research* (2011). pages 6

[10] Giulio Garbi, Emilio Incerto, and Mirco Tribastone. 2020. Learning Queuing Networks by Recurrent Neural Networks. *arXiv e-prints*, Article arXiv:2002.10788 (Feb. 2020), arXiv:2002.10788 pages. arXiv:2002.10788 [cs.PF] pages 1

[11] G.Sh.Tsitsiashvili and M.A.Osipova. 2009. Parameter Estimation for Product-Form Distributions of Queueing Networks. *Probl Inf Transm* 45 (2009), 400–405. pages 1

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *arXiv e-prints*, Article arXiv:1512.03385 (Dec. 2015), arXiv:1512.03385 pages. arXiv:1512.03385 [cs.CV] pages 5

[13] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. [n.d.]. Lecture 6a: Overview of mini-batch gradient descent. *University of Toronto* ([n. d.]). pages 6

[14] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* 9 (12 1997), 1735–80. https://doi.org/10.1162/neco.1997.9.8.1735 pages 3

[15] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR* abs/1502.03167 (2015). arXiv:1502.03167 http://arxiv.org/abs/1502.03167 pages 6

[16] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *3rd International Conference for Learning Representations 2015* (2014). arXiv:1412.6980 [cs.LG] pages 6

[17] Thomas G. Kurtz. 1970. Solutions of Ordinary Differential Equations as Limits of Pure Jump Markov Processes. *Journal of Applied Probability* 7 (1970), 49–58. pages 1

[18] J. F. Pérez and G. Casale. 2017. LINE: Evaluating Software Applications in Unreliable Environments. *IEEE Transactions on Reliability* 66, 3 (2017), 837–853. pages 6

[19] M. Raeis, A. Tizghadam, and A. Leon-Garcia. 2019. Predicting Distributions of Waiting Times in Customer Service Systems using Mixture Density Networks. (2019), 1–6. pages 1

[20] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. 2015. Evaluating approaches to resource demand estimation. *Performance Evaluation* 92 (2015), 51 – 71. https://doi.org/10.1016/j.peva.2015.07.005 pages 1

[21] Charles Sutton and Michael I. Jordan. 2010. Bayesian inference for queueing networks and modeling of internet services. *arXiv e-prints*, Article arXiv:1001.3355 (Jan. 2010), arXiv:1001.3355 pages. arXiv:1001.3355 [stat.ML] pages 1

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762 pages 2, 4, 5

[23] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. 2019. Learning Deep Transformer Models for Machine Translation. *arXiv e-prints*, Article arXiv:1906.01787 (June 2019), arXiv:1906.01787 pages. arXiv:1906.01787 [cs.CL] pages 4

[24] Weikun Wang, Giuliano Casale, Ajay Kattepur, and Manoj Nambiar. 2016. Maximum Likelihood Estimation of Closed Queueing Network Demands from Queue Length Data. *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (2016), 3–14. pages 1

[25] Weikun Wang, Giuliano Casale, and Charles Sutton. 2016. A Bayesian Approach to Parameter Inference in Queueing Networks. *ACM Transactions on Modeling and Computer Simulation* 27, 2 (2016). pages 1